NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

MULTIPLEXING ETHERNET IN A MULTI-USER
CP/M-86 SYSTEM

by

Izzet Percinler

June 1984

Thesis Advisor: Uno R. Kodres

Approved for public release; distribution unlimited

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)

Multiplexing Ethernet in a Multi-user CP/M-86 System

5. TYPE OF REPORT & PERIOD COVERED

Master's Thesis
June 1984

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)

Izzet Percinler

8. CONTRACT OR GRANT NUMBER(s)

9. PERFORMING ORGANIZATION NAME AND ADDRESS

Naval Postgraduate School
Monterey, CA 93943

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS

Naval Postgraduate School
Monterey, CA 93943

12. REPORT DATE

June 1984

13. NUMBER OF PAGES

121

14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)

15. SECURITY CLASS. (of this report)

Unclassified

15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Local area network; Ethernet; Multiplexing; Single board computers; CP/M-86

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This thesis describes the Data Communications Software that demonstrates the viability of multiplexing Intel iSBC 86/12A Single Board Computers contained in a Multibus-based multi-user CP/M system. The NI3010 MULTIBUS-ETHERNET Communication Controller Board provides the interface between Multibus-based Microcomputers and an Ethernet Local Area Network. The Intel MDS (CP/M-86 based), for demonstration purposes within the context of this thesis, acts as a remote host. In future applications, it is envisioned that the remote host(s) will be either MDS-based systems or Digital Equipment Corporation's (DEC) VAX-11/780 (Unix Operating System), or the IBM 3033 mainframe.

Approved for public release; distribution unlimited.

Multiplexing Ethernet in a Multi-user CP/M-86 System

by

Izzet Percinler
Major, Turkish Army
B.S., Turkish Army Academy, 1967
M.S., AITIA Statistics , 1977

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1984

ABSTRACT

This thesis describes the Data Communications Software that demonstrates the viability of multiplexing Intel iSBC 86/12A Single Board Computers contained in a Multibus-based multi-user CP/M system. The NI3010 MULTIBUS-ETHERNET Communication Controller Board provides the interface between Multibus-based Microcomputers and an Ethernet Local Area Network. The Intel MDS (CP/M-86 based), for demonstration purposes within the context of this thesis, acts as a remote host. In future applications, it is envisioned that the remote host(s) will be either MDS-based systems or Digital Equipment Corporation's (DEC) VAX-11/780 (Unix Operating System), or the IBM 3033 mainframe.

DISCLAIMER

Many terms used in this thesis are registered trademarks of commercial products. Rather than attempting to cite each individual occurrence of a trademark, all registered trademarks appearing in this thesis are listed below, following the firm holding the trademark:

Digital Equipment Corporation, Maynard, Massachusetts

VAX-11/780

UNIX Operating System

Digital Research Incorporated, Pacific Grove, California

CP/M-86 Operating System
PL/I-86 Programming Language
RASM 86 Assembler

Intel Corporation, Santa Clara , California

8086 Microprccessor
Multibus Bus Architecture
Intel SBC 86/12A
Intel MDS

INTERLAN , Inc. (Chelmsford, Mass)

NI3010 Multibus Ethernet Communications Controller Board

Xerox Corporation, (Stamford, Connecticut)

ETHERNET

XEROX-DIGITAL-INTEL

ETHERNET Version 1.0

ETHERNET Version 2.0

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENT

This thesis is devoted to the memory of my sister, Muazzez PERCINLER who passed away so early.

Special thanks to my wife Jale and our daughters Ece and Nil, for their patience during the heavy load of the education period.

Special thanks to my Thesis Advisor Professor Uno R. KODRES for his guidance and stimulating my interests in the challenging development of the Data Communications Software pertinent to this Thesis.

Special thanks are extended to LT David J. BREWER, USN for his invaluable assistance and contributions, especially in the area of Local Area Network concepts. His concurrent development of ETHERNET Communications Software, in support of the Extended MCORTEX Operating System, provided me with insight and inspiration.

I. HISTORY AND INTRODUCTION

A. BACKGROUND

It became very evident to the early designers and manufacturers of computer systems and components that some methodology must be developed to share the expensive elements (within the system) among various users. Among the earliest and easiest methods was that of "batch processing." Batch processing techniques became fairly sophisticated, through the use of, for example, IBM's Job Control Language (JCL). However, only until "timesharing" systems were developed did the full benefits of the sharing of limited resources became evident. Sophisticated batch and timesharing system designers of yesteryear were very concerned with properly utilizing their expensive resource, the Central Processing Unit (CPU).

Contemporary computer system architects are able to benefit from the rapid technological advances in VLSI techniques. As a result, the Single Board Computers (SBC) and Single Chip Computers (SCC) are inexpensive, relative to hard disks, printers, graphic devices, and other peripherals. The fundamental question posed to these architects is: How can these peripherals "appear" less expensive to the users? The fundamental answer is: Distribute the resources among the users, thus reducing the cost per user.

The use of Local Area Networks (LAN) is the most economical means by which limited (expensive) resources can be distributed and utilized by a volume of users. An incremental increase in processing power is easily realized through the replication of inexpensive SBC's within a node or host, and reliability is an almost fortuitous consequence.

B. AEGIS

The AEGIS Weapons System Simulation Laboratory at NPGS Monterey, supports concentrated efforts in a continuing feasibility study of replacing the present 4-bay AN/UYK-7 based system, installed aboard the U.S. Navy's newest guided missile cruisers (CG-47 class). The Simulation laboratory group is exploring multiple SBC architectural implementations for:

1. Increased Performance,

2. Increased Reliability,

3. Increased Survivability

by connecting clusters into a LAN system.

C. PURPOSE

The main purpose of this thesis is to develop data communications software between the AEGIS Laboratory's multi-user CP/M system and the Intel MDS. This is a stepping stone to the VAX and IBM virtual terminal idea. The microcomputers can be at one moment workstations, which edit source code and the next moment the source code can be transferred to IBM/VAX hosts. Compilers (e.g. ADA cross-compiler) can translate the source code and results can be sent back for execution and testing.

Details concerning the multi-user CP/M-86 Operating System may be found in [Ref. 1]. The CP/M-86 Operating System for the MDS is mentioned in [Ref. 2]. Basically, each single board computer and the MDS uses a single-user version of CP/M-86.

With the proper data communications software, message transfer between multi-microcomputers and MDS host system is not only possible, but can be accomplished efficiently.

Interlan's NI3010 MULTIBUS-ETHERNET Communications Controller Board is used in both the MDS system (Multibus-based) and the multi-user CP/M-86 system.

Thus, each Intel iSBC 86/12A is going to behave like a virtual terminal of the MDS system. This thesis demonstrates the viability of the data communications between multi-microcomputers and the host system on the Ethernet Local Area Network.

The International Standard Organization's (ISO) Open Systems Interface (OSI) 7-layer architecture model is used as a developmental guide in order to ensure the compatibility of future implementations and ease of integration into existing data communications systems.

This thesis gives the History and Introduction in Chapter I. Local Area Networks, primarily ETHERNET, is described in Chapter II. System Hardware is mentioned in Chapter III. System Software is given in Chapter IV. System Design and Implementation is described in Chapter V. Results and Conclusions are given in Chapter VI. Data Communication Software developed for this implementation is illustrated in Appendixes A - J.

The purpose of this thesis is to construct a software interface to the CP/M-86 Operating System so that messages can be transported between the iSBC's and MDS systems via the Ethernet Local Area Network. By using the Data Communications Software, each iSBC 86/12A (i.e. , each user in the multiuser system) can :

1. Send Messages to MDS .

2. Receive messages from MDS.

II. LOCAL AREA NETWORKS

A. GENERAL

The data communications within a building or a complex of buildings can be realized through the use of Local Area Networks (LAN). LAN's span distances between several meters through several kilometers in length. The speed of data transmission of a LAN is typically 100 Kbps to 10 Mbps. The transmission media is generally inexpensive relative to the cost of the system being supported by it. The transmission medium can be twisted pair, coaxial cable, or fiber optics. The necessary hardware consists of the interface units for host computers, the transmission medium, and a transmission control mechanism over the medium.

The software protocols make the system work in the desired way. These are implemented in the host computers and provides the control of data transmission through hardware components. The ISO/OSI 7-layer model's lower levels -Physical and Data Control Layers- are strictly implemented and the higher levels - Network, Transport, Session, Presentation, and Application layers - implementation depends mostly on Long-haul packet communication networks.

Local Area Networks carry information usually via broadband, baseband, or twisted pair media. Broadband and baseband are the terms that describe different varities of coaxial cable used for local area networks. Actually this is misleading, since broadband and baseband are signalling techniques which are independent of the physical medium.

1. Broadband

Broadband signalling techniques generally allows the transmission of data over longer distances. Broadband cable is standard 75-ohm cable. In broadband local area networks the capacity is used to produce a large number of frequency subchannels from one physical channel. The transmission rates on broadband networks are less than the rates on baseband networks.

2. Baseband

Baseband networks are more geographically limited to transmit digital data only. The data rate is higher in baseband networks, but is limited to one channel. The data rate is as much as 10 Megabits per second for distances up to one mile. Baseband cable is 50-ohm coaxial cable and is generally more fragile than broadband. Baseband systems provide a control scheme to allow data to be sent without interference from other stations.

Several schemes have been used including simple time-division multiplexing or "slot" concepts such as the Cambridge ring. Most modern baseband networks use either a contention system, CSMA/CD (Carrier Sense Multiple Access with Collision Detection), or a rotating-control method called Token Passing. CSMA/CD networks listen for conflicting traffic to avoid data collision, while token passing networks circulate a token to permit a station to capture available transmission time.

3. Twisted Wire

The twisted wire is the least expensive transmission medium. Its capacity is limited and implementation is very limited in geographic range. Twisted wires provide top transmission rates of 1 million bits per second at distances up to 4,000 feet if a line driver is used. They are highly susceptible to electical interference, which can often scramble data transmissions.

4. Optic Fiber

The optic fiber transmits data at very high rates and is extremely secure. It is less bulky than cables and twisted wires. It has the necessary qualifications to be a candidate for future transmission media. Fiber optic cables are not susceptible to electromagnetic radiation. Thus problems such as ground loops, crosstalk, and lightning interference are eliminated. No electrical signals are transmitted between equipments interconnected by the glass fibers, thereby eliminating the possibility of electrical surges or short circuits. Morever, it is almost impossible to tap into the Ethernet data bus without immediate detection, a security advantage over coaxial ones. With proprietary collision detection technology, greater bandwidth, and lower attenuation of optical fibers, data can be transmitted between nodes separated by 2.5 KM. , without repeaters. This is in contrast to 0.5 Km. of coaxial-cabled Ethernet. "FIBER OPTIC/NET ONE" is an example of a fiber optic connected Ethernet Communication System manufactured by Ungermann-Bass (Santa Clara, California) .

B. TOPOLOGIES

The common Local Area Network topologies include the point-to-point, star, ring, and bus topologies as shown in Figure 2.1 [Ref. 3]. Small networks linked with twisted wires often use the point-to-point configuration. A single wire connects each host in the network. There is no need to provide a central communications controller and is normally applied only for a small number of nodes.

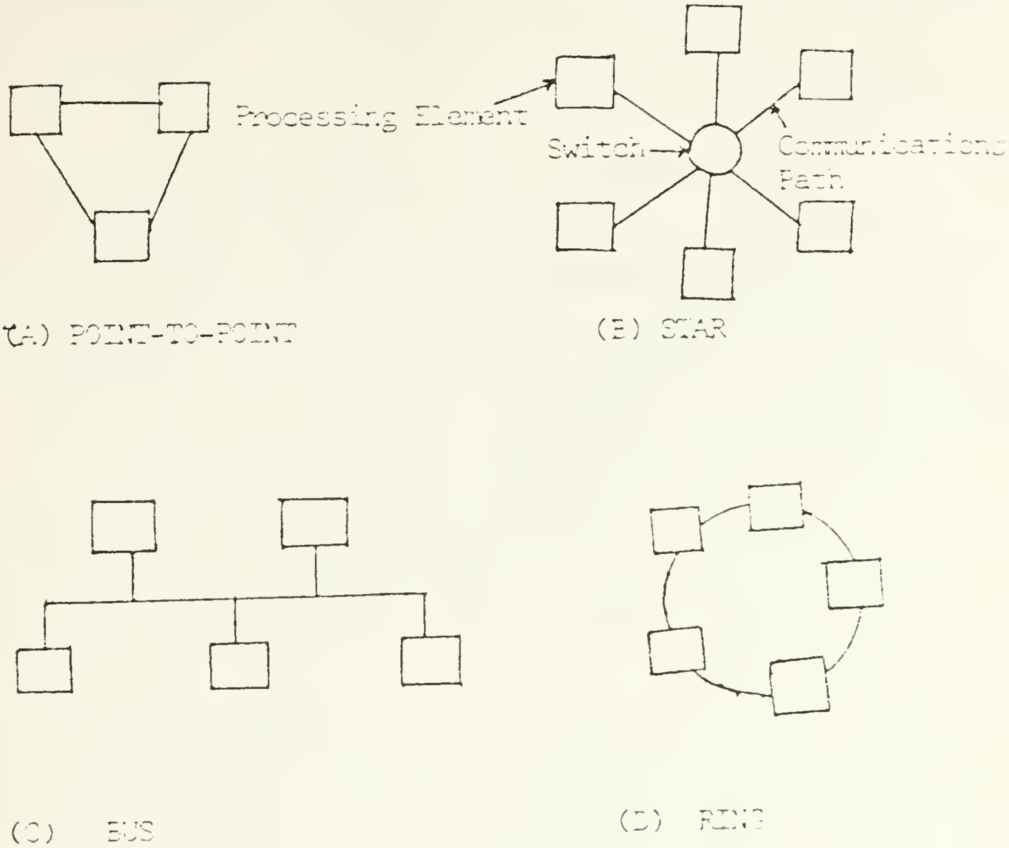

Figure 2.1 Local Area Network Topologies.

In a star configuration, each host on the network connects to a central controller which performs all the switching. The central controller manages data transfer. It is vulnerable when any problem exists in the central controller.

In the bus topology, hosts directly connect to a central cable that runs the length of the network. Each host has a unique address, to receive its mail. The software data communications protocols manage data communication.

The ring configured topology can be visualized as a bus network with two ends tied together. The data transmission is unidirectional along predefined transmission routes. A single channel ring network is vulnerable to a failure in the connecting cable or a retransmission device.

C. ETHERNET

The original design of the Ethernet system is due primarily to Robert M. Metcalfe and David R. Boggs [Ref. 4].

The coordinated efforts of DIGITAL, INTEL and XEROX on the XEROX's experimental Ethernet produced a well specified Ethernet standard describing Physical and Data Link Control layers in detail in 1982 [Ref. 5].

Ethernet is the Local Area Network developed by XEROX in 1975 at the XEROX Corporation's Palo Alto Research Center.

Ethernet designers introduced both the "Listen Before Talk" and "Listen While Talk" mechanisms into Ethernet. "Listen before Talk" method is also known as Carrier Sense Multiple Access, in which stations monitor the channel prior to transmission. In this method, stations wait for a statistical period before retransmission in the event of a collision. This method yields utilization in excess of 80 % capacity. With the second method "Listen While Talk" collisions should occur only when two or more stations find the channel silent and begin transmitting simultaneously. With a "Listen While Talk" mechanism, colliding packets can very rapidly be truncated so as not to waste an entire packet time on the channel. The use of this mechanism yields utilization in excess of 90 % of channel capacity.

Since Ethernet uses CSMA/CD, every host competes for access to the one channel on the Ethernet cable. CSMA portion of the access mechanism resides in the NIU or integral board on the host system. It determines whether or not the channel is open and either holds the packet in a buffer until the channel is free or transmits the information.

If the Ethernet is overloaded and a collision of packets occurs, the collision detection mechanism (CD), located in the transceiver, observes this through a change in the electrical state of the channel. The CD will automatically "back off" communications on the channel and store the data in a buffer. After an arbitrary wait, the NIU or controller board will retransmit the data. Retransmission is accomplished by using the Truncated Binary Exponential Backoff Algorithm.

The Ethernet is configured by connecting a number of independent terminals through an interface to a transceiver, which is in turn connected to the transmission medium, typically a coaxial cable. The topology of Ethernet is that of an unrooted tree, in the sense that there is a unique path between every pair of stations. Stations can attach to the cable at any point, and the cable can be extended from any of its points in any direction by the use of repeaters.

Speed conversion between stations is intrinsic to the Ethernet interfaces, since all transmission within the network takes place at a speed different from the terminals. Flow control must be exercised in the interface to prevent buffer overflow with resultant loss of data.

The original Ethernet designers distinguished between the interface (responsible for serializing and deserializing the parallel data used by the station, for computing and checking the Cyclic Redundancy Checksum, and for accepting only those packets addressed to the station it serves) and the Controller (responsible for retransmitting colliding or unacknowledged packets) . In their implementation, the interface was designed separately for each type of station, but the controller resided in the station itself (generally as low-level firmware or software) . Subsequent implementations of Ethernet have taken the approach of combining the interface and controller functions into a separate, buffered device between the station and the Ethernet transceiver.

Ethernet is a branching broadcast communication system for carrying digital data packets among locally distributed computing stations. The packet transport mechanism provided by Ethernet has been used to build systems which can be viewed as either local computer networks or loosely coupled multiprocessors. An Ethernet's shared communication facility, its Ether, is a passive broadcast medium with no central control. The coordination of access to the Ether for packet broadcast is distributed among the contending transmitting stations using controlled statistical arbitration. The switching of packets to their destinations on the Ether is distributed among the receiving stations using packet address recognition.

The Characteristics of Ethernet are :

Data Rate : 10 Megabits per second

Transmission Medium : Baseband Coaxial Cable

Maximum Number of Nodes : 1,024

Topology : Linear Bus

Access Method : CSMA/CD

Network Segment : 500 Meters (1,600 feet) per segment

The Maximum distance of an Ethernet with repeaters : 2.5 Km.

The max. no. of transceivers to be connected to a single segment : 100

The shortest distance between the transceivers : 2.5 Meters.

The Ethernet Packet format size ranges from 64 Bytes (Minimum) to 1516 Bytes (Maximum). The difference depends on the size of data field .

The minimum data field size is 46 data bytes. The maximum data field size is 1500 Bytes. Each packet starts with a Preamble field, which is an 8 Byte Synchronization pattern containing alternating 1's and 0's and ending with two consecutive 1's. The Destination Address is 6 Bytes long. It specifies the station(s) to which the packet is being transmitted. If the first bit is 1, all stations are addressed by this broadcast address. The source address is 6 bytes and shows the sender station. The type field is 2 bytes and is used for Gateway purposes. Data Field varies

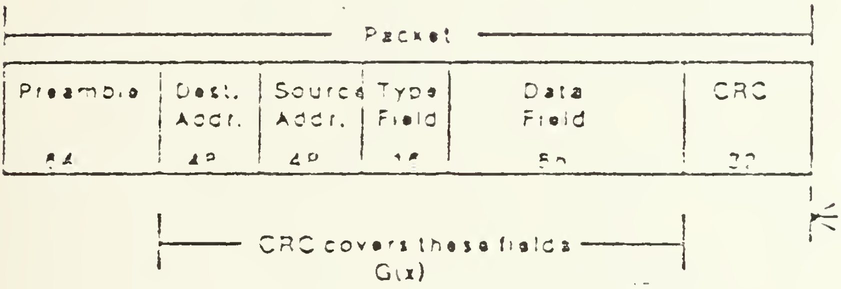

Figure 2.2 The ETHERNET Packet Format.

from 46 to 1500 bytes. Refer to Figure 2.2 [Ref. 6]. Packet check sequence is 4 bytes and contains a Cyclic Redundancy

Check (CRC) code. The Ethernet Efficiency depends on the packet size and the number of nodes connected to LAN.

D. FUTURE OF LAN'S

The electronic mail systems and sharing of files and peripherals in the age of microcomputers make LAN's a necessity. Ethernet, in spite of varying degrees of resistance will be more attractive in the future due to the availability of inexpensive VLSI Ethernet chips. The fiber optic technology is very suitable for military application where secure transmission is highly desirable and where a predictable response time is necessary. Fiber optic ring networks which use a token passing access control are typically suggested for military applications.

III. HARDWARE DEVICES

A. INTERLAN'S NI3010 MULTIBUS ETHERNET COMMUNICATIONS CONTROLLER

The NI3010 is the hardware essence of this thesis. It is a single board that along with a transceiver provides a host MULTIBUS system with a complete connection to an Ethernet network. Figure 3.1 shows the ETHERNET architecture and the NI3010 implementation [Ref. 7].

The NI3010 is a DMA (Direct Memory Access) device that responds to commands issued by the host MULTIBUS System. It incorporates Interlan's NM10 ETHERNET protocol module and complies in full with the Xerox/Intel/Digital Ethernet Specification, Version 1.0. It performs the specified data link and physical channel functions [Ref. 8].

NI3010 consists of the Interlan MULTIBUS Interface Board (MIB) and the Interlan NM10 ETHERNET Protocol Module. The MIB contains the logic necessary for transferring data between the NM10 and the host MULTIBUS system. NM10 is the Interlan's ETHERNET Protocol Module. It contains the data communications logic that interfaces the MIB to the ETHERNET. Data travel to and from the MIB through an 8-bit bidirectional data bus to internal memory buffer registers. Transmit data then enters a transmit buffer and awaits transfer to the Ethernet. Receive data from the Ethernet enters a receive buffer and awaits transfer to host MULTIBUS memory. The Ethernet architecture and NI3010 Implementation is described in Stotzer's thesis [Ref. 9].

The NM10 contains two FIFO (first_in, first_out) buffers:

1. A 2 KByte TRANSMIT BUFFER:

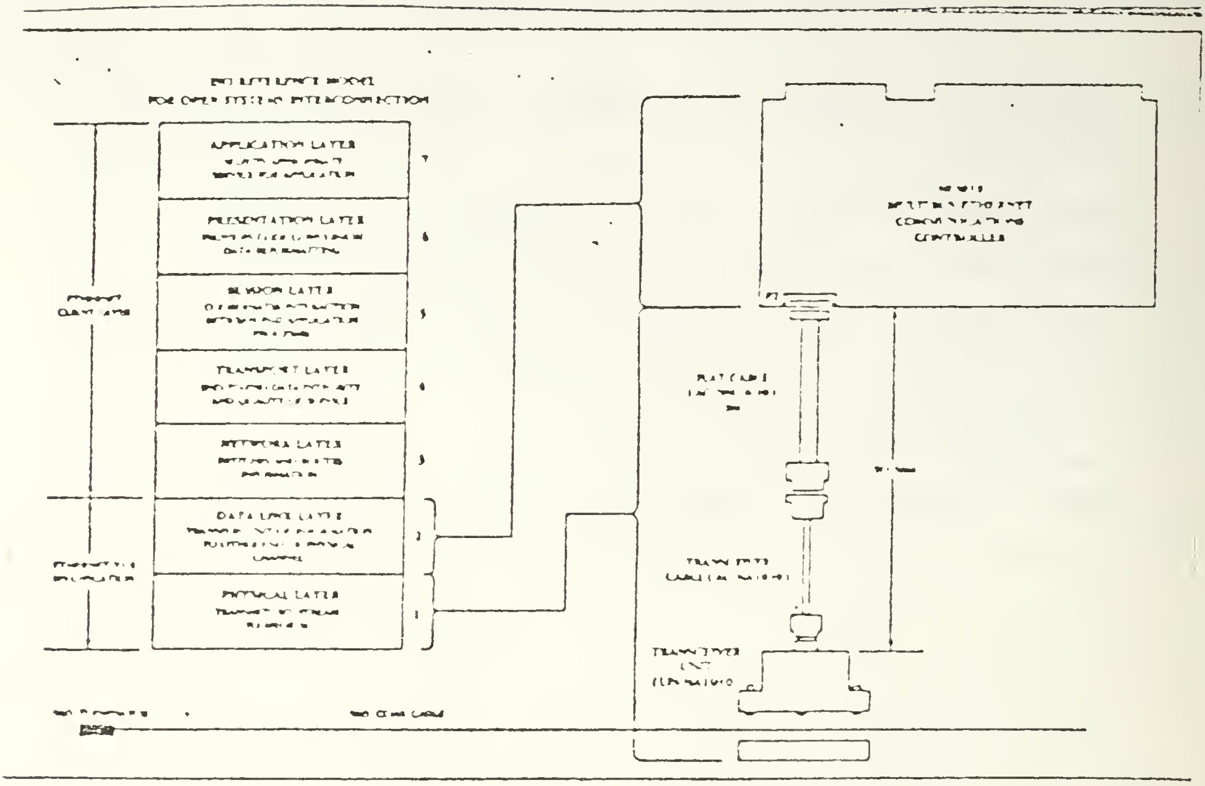

Figure 3.1   ETHERNET Architecture and NI3010 Implementation.

This buffer allows the host to transfer each transmit packet to the Ni3010 only once, independent of network traffic. To send the data out on the Ethernet, the host must issue a Load Transmit Data and Send Command (29H) to the NI3010. If a network access collision occurs, the NM10 automatically reschedules transmission. Because the transmit frame is still available in the transmit buffer, the host need not to send it again to the NI3010.

2. A 16 KByte RECEIVE BUFFER:

This buffer stores receive frames. It buffers the Multibus from the unpredictable arrival times of network traffic, consequently reducing the time-critical service requirements on the host Multibus system.

Transmit Data Block in MULTIBUS Memory: The host transfers data to the NI3010 by setting up a transmit block in its own memory, writing the NI3010's bus address registers (BAR) with the block's starting address, writing the NI3010's byte count registers (BCR) with the block's byte count, and then initiating a transmit DMA Operation. The host must set up the transmit block in a particular format, as shown in Figure 3.2 [Ref. 7].

Only one frame can be loaded in the NI3010's transmit FIFO at a time. When a user issues a load transmit data and send command (29H), the NI3010 transmits all the data in its FIFO as an Ethernet frame. It adds the Ethernet preamble, the source address, and the CRC value.

Receive Data Format in Multibus Memory :

When the host receives a receive_block_available interrupt from the NI3010, it reserves a block in its own memory for the receive data, writes the NI3010's bus address registers (BAR) with the block's starting address, writes the NI3010's byte count registers (BCR) with the block's byte count, and then initiates a receive DMA. The receive data enter host memory in the particular format as shown in Figure 3.3 [Ref. 7].

When the NI3010 receives an Ethernet frame, it strips off the preamble and stores the rest of the frame in its receive FIFO. The rest of the frame includes 6 bytes of destination address, 6 bytes of source address, 46 to 1,500 bytes of data, and 4 bytes of CRC.

```
                          7                          0
                         +----------------------------+
      BAR + 0    -->     | destination address (A)    |
                         |----------------------------|
                         | destination address (B)    |
                         |----------------------------|
                         | destination address (C)    |
                         |----------------------------|
                         | destination address (D)    |
                         |----------------------------|
                         | destination address (E)    |
                         |----------------------------|
                         | destination address (F)    |
                         |----------------------------|
                         |      type field  (A)       |
                         |----------------------------|
                         |      type field  (B)       |
                         |----------------------------|
                         |     data (first byte)      |
                         |----------------------------|
                         |             .              |
                         |             .              |
                         |             .              |
                         -                            -
                         |                            |
                         |----------------------------|
  BAR - BCR - 1    -->   |     data (last byte)       |
                         +----------------------------+
```

Figure 3.2   Transmit Data Block in Multibus Memory.

When the NI3010 transfers a frame to the host, it adds 4 bytes of header. The first byte is the frame status; the second is a null byte; and the third and fourth are the frame length. The frame length is a binary value representing the number of bytes in the received frame.

When the host initiates a receive DMA, it receives the oldest receive frame stored in the NI3010's receive FIFO. After receiving that frame, the host once again enables a

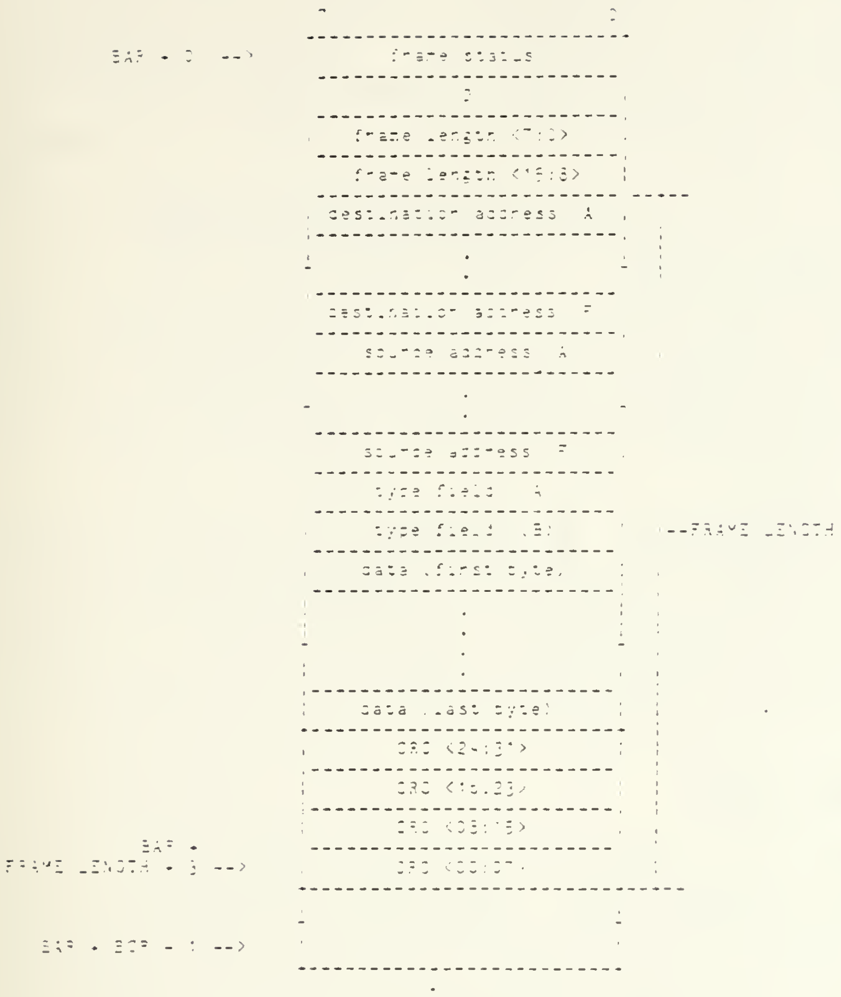

Figure 3.3 Receive Data Block in Multibus Memory.

receive_block_available interrupt. If the NI3010 has another frame ready, it interrupts again.

B. SINGLE BOARD COMPUTERS

1. General

A single board microcomputer is a single printed circuit board containing as a minimum, a processor, memory

(ROM and RAM) and input/output ports. It usually has a combination of serial and parallel ports. It may also include a counter/timer function and a bus interconnection scheme. An SBC family may also include other functional elements (such as memory and I/O functions) on circuit boards of the same format as the microcomputer board. Single Board Microcomputers are sometimes called "monoboard" microcomputers. In January 1984, Intel combined a single-chip microcomputer and data communications capabilities with industry-standard networking software in its iSBC 186/51 Single Board Communications Computer and INA 960 Local Area Networking software.

Many manufacturers are producing single board computers. The software has matured and most systems come fully supported with high level languages PL/I, PASCAL, FORTRAN, COBOL. The first SBC's appeared in 1976. The SBC's capabilities are evolving rapidly.

2. Intel iSBC 86/12A

The Intel iSBC 86/12A has an Intel 8086 CPU, 32 or 64 Kbytes RAM, 0-32 Kbytes ROM, MULTIBUS interface control logic, serial interface, Intel 8255 supplying 3 programmable parallel I/O ports, Intel 8253 Programmable Timer (PIT), and an Intel 8259A Programmable Interrupt Controller (PIC). Additionally it has a 16 bit word size, 20 bit address lines (providing 1 Mbytes addressable memory space), 5 Mhz clock frequency, 38.4 Kbaud maximum I/O rate, and has multiprocessing capability. It supports among others FORTRAN, PASCAL, PL/M, PL/I-86, ASSEMBLY Languages and the CP/M-86 Operating System.

a. Microcomputer CPU

INTEL 8086 has the seven 8-bit registers of the 8080, with eight added registers so that the registers can be paired up to form four 16-bit registers. The 16-bit registers are AX, BX, CX, and DX: when considered as 8-bit registers, they are AL, AH, BL, BH, CL, CH, DL, and DH (L = Low-order byte ; H = High-order byte). Refer to Figure 3.5 [Ref. 1].

3. Multi-Microcomputers

Systems built around microprocessor-based information stations will soon rival the speed, power, and capacity of some mainframes. Advances in VLSI technology is helping by developing monolithic chips. Adding more terminals to a uniprocessor decrease the power, speed, and capacity of individual users. However in multi-microcomputer applications more microcomputers provide more parallel computing power. Given the right multiprocessing configuration, a linear increase in performance can result by adding another microcomputer [Ref. 10]. The rapidly dropping cost of microprocessors will help to increase this kind of application in the future.

Although the concept of multiprocessing is not new, the concept of implementing such a scheme using microprocessors is. In the 1960's, conventional, relatively expensive mainframe CPU's were connected to a common memory, making it uneconomical to have more than a few processors. Since 1975, considerable effort has been devoted to resolve the problems that limited the practicality of multi-microprocessor systems. These problems include system inefficiencies, interconnection structures, and software system structures. The answers have been sought in the context of closely coupled multi-microprocessor schemes, such as Carnegie-Mellon University's CM* modular multi-microprocessor system.

Coupling involves the degree of interaction between processors. In a loosely coupled system, each processor has

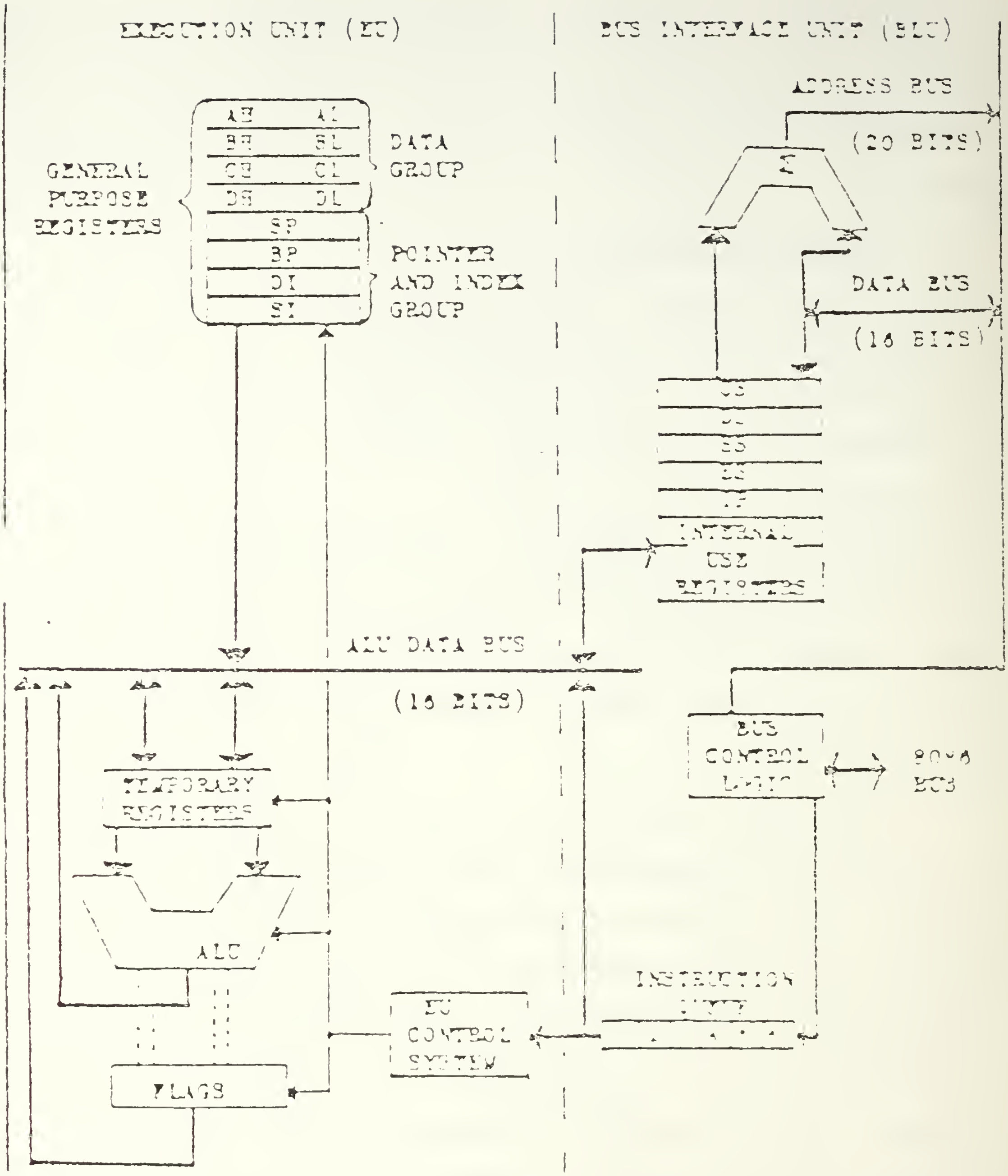

Figure 3.4 The Internal Architecture of Intel 8086.

some dedicated complement of program and data memory, and possibly some I/O devices. Each such unit, which can function independently, communicates with another, using either data transfers over a common bus or point-to-point communication lines or coordinated signals maintained in a commonly accessible store. A tightly coupled system involves multiple processors sharing common program and data memory and I/O transfers. All processors access common buses to perform memory and I/O transfers.

IV. SYSTEM SOFTWARE

A. CP/M-86 OPERATING SYSTEM

CP/M operating system's organization separates hardware-dependent functions in the BIOS from hardware-independent functions in the BDOS and CCP. To move CP/M to a new hardware environment, only the modification of BIOS is necessary. The alteration of the CP/M-86 Operating System for the AEGIS project is described by Candalor and Almquist [Ref. 2]. and [Ref. 11].

BDOS interfaces with the user's programs and does not really change from machine to machine (similar machines). It allows a program to address logical devices (such as drive A or B) so that the program does not need to consider the physical details of how those devices actually work. The BIOS resides at the bottom part of the operating system and translates this logical operating system schema into an actual detail needed to operate the hardware. Therefore, only BIOS must be changed to accommodate different computers.

B. MCORTEX OPERATING SYSTEM

MCORTEX is the real time executive for a multiple processor system used for the SPY-1A Radar Emulation. It is very specialized manager of concurrent processes. For previous work on Mcortex refer to Rowe [Ref. 12].

C. PROGRAMMING LANGUAGES SUPPORT

1. PL/I-86

PL/I-86 has added enhancements to optimize use of the 8086's larger word-size, instruction set, and memory addressing range.

PL/I supports scientific, data processing, text processing, systems programming applications. PL/I was first implemented in 1965 by IBM .

The syntactic structure and dynamic storage allocation features of ALGOL, the record structures and input-output of COBOL, the Arithmetic capabilities of FORTRAN, some string processing, list processing interrupt-trapping features are all combined in PL/I.

PL/I is the most powerful high level language to be used in microcomputers. PL/I is extensively used as a systems development language in the microcomputer industry. PL/I-86 incorporates all the features of PL/I-80 in order to maintain compatibility and adds other enhancements intended to optimize use of the Intel 8086 processor. PL/I is preferred as a high level language in this implementation due to its power and flexibility in microcomputer systems.

PL/I Subset G was standardized in 1979 as an attempt to overcome "the language functionality/memory space-execution time" trade-off that PL/I had suffered in its long and winding history. Subset G has the best features of PL/I.

2. RASM86

RASM-86 processes an 8086 Assembly Language source file in three passes and produces an 8086 machine language object file. RASM-86 can optionally produce three output files :

a. List file (filename.LST),

b. Object File (filename.OBJ),

c. Symbol file (filename.SYM)

from one source file (filename.a86).

The list LST file contains the assembly language listing with any error messages. The object OBJ file contains the object code in Intel 8086 relocatable object format. The symbol SYM file lists any user defined symbols. The three files have the same filename as the source file.

D. DATA COMMUNICATIONS SOFTWARE

The following descriptions of the functional modules of the data communication software is provided as a quick reference to describe the modules in limited detail. A more descriptive discussion in the code listings is provided in Appendixes A - J.

1. "remote.pli"

This main module, after being linked to "multmods", "cmaccess", "interrpt", and "sync" assembler object files, provides the essense of data communications software between multiple single board computers and a remote host. The resultant transmit command file is "remote.cmd". Figure 4.1 shows how the "remote.cmd" program is generated. In this implementation iSBC86/12A's and an Intel MDS host computer are used. The same program is run on each SBC. It is provided in Appendix A.

a. "sync.a86"

It provides synchronization of CP/M-86 users (through program "remote") requesting service of the

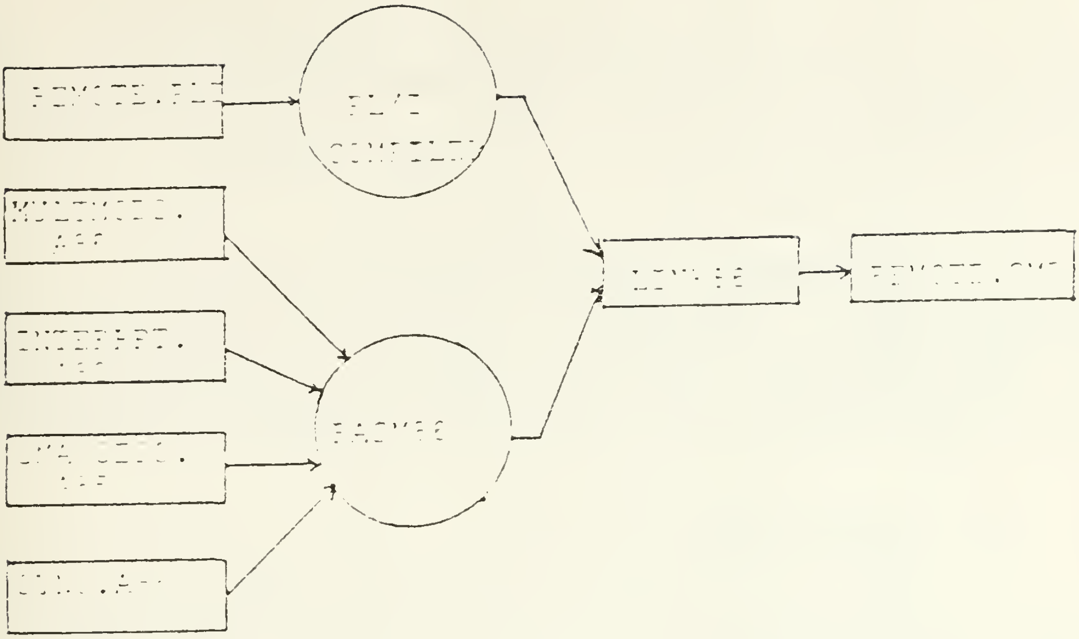

Figure 4.1 "remote.cmd" Program Generation.

Ethernet. A slightly modified version of the Ticket/Server system used in Almquist and Steven's thesis is used here. This module contains the code which prevents more than one computer from accessing shared resources (the NI3010, in this case) while another user may be already issuing commands to the device. The program is provided in Appendix B.

c. " multmods.a86 "

This module is used to load DMA address and byte count registers, and write to specific I/O ports. (write_io_port and read_io_port). It is provided in Appendix C.

c. "cmaccess.a86"

This module moves data from local memory to common memory and vice versa until data bytes are exhausted. It is provided in Appendix D.

d. "interrpt.a86"

This module provides the initialization of CPU interrupts, and it enables and disables CPU interrupts. The high level interrupt handler routine of the "remote.pli" module is also called from here. It is provided in Appendix E.

e. "NI3010.DCL"

This gives the I/O port addresses and interrupt enable status register values and command function codes. These are specific to the use of Interlan NI3010 MULTIBUS to ETHERNET interface. It is provided in Appendix F.

2. "remote5.cmd"

This program is run on the MDS host computer and sends and receives packets via ETHERNET to act as a distant host to the multi-user CP/M-86 system. Figure 4.2 shows the generation of "remote5.cmd": It is provided in Appendix G. The program is a modified version of the main module.

3. "r5mod.pli"

This module is designed to send and receive packets via ETHERNET to act as a distant host to the multi-user CP/M system for performance metrics purposes. The generation of "r5mod.cmd" is analogues to that of "remote5.cmd", as illustrated in Figure 4.2 .

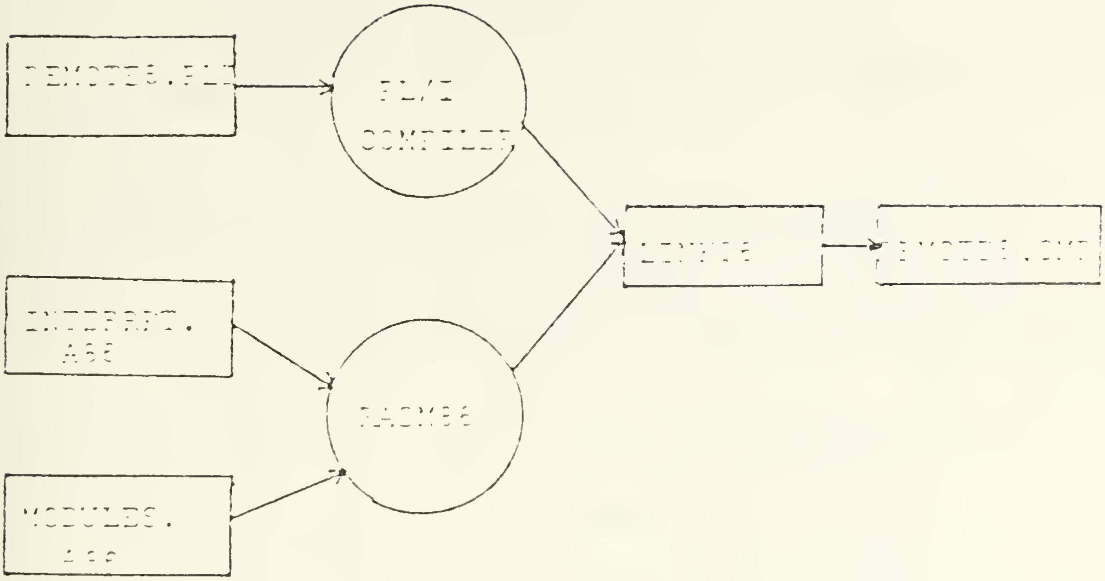

Figure 4.2 The "remote5.cmd" generation.

4. "ether.cmd"

This routine (transient command) must be invoked any time a hard reset occurs on the Multibus backplane. It reinitializes common memory to the point at which the first user of the Ethernet services will now do a full reinitialization of synchronization variables and also place the NI3010 board on line. It is provided in Appendix I.

5. "tstether.pli"

This module is a modified version of "remote.pli" and is designed to function as a test program of the data communications software to demonstrate and analyze the speed of data transferred via Ethernet. The generation of "tstether.cmd" is analoguous to that of "remote.cmd", as illustrated in Figure 4.1 . It is given in Appendix J.

V. SYSTEM DESIGN AND IMPLEMENTATION

A. ISO REFERENCE MODEL FOR OPEN SYSTEMS INTERCONNECTIONS

1. General

The International Organization for Standardization has developed a Reference Model of Open Systems Interconnection (ISO-OSI) comprised of seven layers. These layers are :

a. The PHYSICAL Layer

b. The DATA LINK Layer

c. The NETWORK Layer

d. The TRANSPORT Layer

e. The SESSION Layer

f. The PRESENTATION Layer

g. The APPLICATION Layer

2. Layers

a. Physical Layer (Layer 1)

The physical layer deals with transmission of raw bit stream, and the electrical protocols. The physical link layer in a network is responsible for delivering the bits from one node to another. It encompasses plug, pin connections, interface hardware, impedances, the actual transmission medium, the signalling means (voltage or current levels, frequency channels, modulation techniques) , and the data rate.

RS232C Serial Communications interface, MIL-STD-188, RS449, V.24, X.21 bis, X.21, V.35, 303 are examples of PHYSICAL layer Protocols.

Physical links do not guarantee reliable service. Electrical noise in the environment can cause interference. The fiber optics media is not susceptible to this but may have either self-generated errors or external interference in the receiver due to the extremely low signal levels (nano-amperes) in the receiver. Physical link specifies how a transmitter sends bits on the transmission media, and how a given receiver accepts them for its node. In this application the NI3010 provides Layer 1 protocol functions.

b. Data Link Layer (layer 2)

The DATA LINK Layer deals with issues of converting unreliable transmission links into reliable ones by using techniques like checksums to validate information received over the line. Carrier Sense Multiple Access with Collision Detection and Token Passing are two main methods used at this level. IEEE 802 specification is being studied for a standardization.

DATA LINK Layer establishes a communications link between micro and mainframe, manages channel access, and frames data to assure correct sequence and checking of message integrity. DATA LINK Control Protocols establish end connections between hosts and handles retransmission requests and handshaking. This level identifies sending and receiving stations through polling or selection. It also handles functions such as status requests, station reset, restart, start acknowledge, and hangup.

This is the level that solves problems in framing data, that is, deciding which bits are characters and which are messages, and in error control, by detecting data errors, confirming correct messages, or requesting transmission of bad messages. It also numbers the messages to avoid duplication and prevent losses and identifies those messages that are retransmissions. Its last function is line control, or determining which station on a half-duplex or multipoint (network) line transmits and which receives.

The DATA LINK layer does error handling, framing, link management, data link control, information transparency. BSC (Bisync), Start-Stop, HDLC, SDLC, UDLC, BDLC, ADCCP, CSMA, CSMA/CD, TOKEN PASSING, IEEE 802 are examples of DATA LINK Layer Protocols. The NIS010 implements most of Layer 2 functions.

c. Network Layer (Layer 3)

The NETWORK LAYER deals with conventions that govern the transmission of messages over the network. X.25 is being accepted as international standard. Message routing, flow control, message fragmentation and reassembling are some functions of Network Control layer. It addresses and routes the messages. X.21, X.25, Request-Response, Autodial are Network Layer Protocol examples. There are also gateway protocols like X.75, IP, GGP . In this application, data is formed into packets for transmission at this Layer. The "Transmit Packet" procedure of "remote.pli" operates with the specification of this layer.

d. Transport Layer (Layer 4)

The TRANSPORT Layer is used to shield the customer's portion of the network from the carrier's portion; thus a change in carrier should be transparent to the computers at the two ends of the link.

The TRANSPORT Layer controls the Communication session so that data is exchanged reliably and in orderly fashion. TCP, TP are Transport Layer Protocol examples. Since there is no route changing in this application, this layer is not applicable here.

e. Session Layer (Layer 5)

The SESSION Layer deals with setting up, managing, and splitting out process-to-process connection.

The SESSION Control Layer manages connections between the applications processes, setting and controlling systemized aspects of communication such as establishment and termination of connections, end-to-end message unit data control, and dialogue control.

f. Presentation Layer (Layer 6)

The PRESENTATION Layer deals with transformations (like data compression) on the data to be transmitted.

The PRESENTATION CONTROL Layer translates code data and converts it to display formats for terminal screens (or the micro screen), printers, and other peripherals. Data is compacted or expanded, structured for file transfer or for command translation. This layer performs an especially crucial function, since it ensures that data is in user-friendly and transparent forms. Since data transformations is not dealt with, this layer is not addressed.

g. Application Layer (Layer 7)

The APPLICATION Layer refers to the ability of application programs involved in communication to freely exchange data and programs. It supports use and application tasks and systems management such as resource sharing, file transfers, remote file access, database management, and network management. It is the topmost layer of all 7 layers. "remote.pli" operates with the Specifications of this layer. Data exchange is achieved, multiplexing is achieved without bothering the other users transmitting apparently at the same time.

B. DEVICE MULTIPLEXING

1. General

Process multiplexing is a technique for sharing processor resources. The inner traffic controller of an operating system multiplexes the physical processor among a pool of more numerous virtual processors. The traffic controller of an operating system multiplexes virtual processors among a larger number of user processes competing for resources. The user accessible inter-process communication and synchronization primitives (ADVANCE, AWAIT, and TICKET) provided at this level allow the user to easily address complex system-wide inter-process synchronization requirements.

In order to multiplex the one channel NI3010 among multi microcomputers, a synchronization method was developed using a First Come First Served (FCFS) schema.

2. Synchronization Primitives

a. Ticket

The inspiration behind the TICKET operation is the automatic ticket machine that is used to control the order of service in catalogue sales departments. The ticket machine gives out ascending numbers to people as they enter the store, and by comparing the numbers on the tickets one can determine who arrived at the ticket machine first. Furthermore, the person at the counter can serve the customers in order by calling for the customer whose number is one greater than the one previously served, when he is ready to serve a new customer [Ref. 13].

b. Await

Await allows a process to suspend its execution pending the occurrence of a specified event. AWAIT is the operating system's primitive which allows the ticket number to be compared to the service number. If both priority for the process and eventcount >= threshold then the process will be executed. If the service number reaches the value of the ticket number the pending process is ready to proceed. As soon as a processor is available, the process is allowed to continue. The await operations do not terminate until the Advance operation of the producer that got the value t-1 from its ticket operation is executed.

c. Advance

Advance is the operating system's primitive used to increment a given eventcount. The service number is incremented after the server completes its service.

d. Ticket Server Technique

(1) "Call Request". "Call Request" accesses the "Ticket" variable in common memory for a ticket number, increments that variable and waits until the obtained ticket number is equal to the value of the "server" variable, a number also found in common memory.

(2) "Call Release". "Call Release" advances the "Server" number by incrementing it, which in turn releases the shared resource to be used by the holder of the next ticket value.

C. IMPLEMENTATION

1. Design Considerations

The software developed in this implementation is written, to the maximum extent possible, in a High Level Language (HLL), PL/I. The ease of modification/maintenance and readability is a well proven concept and the extensive merits of an HLL shall not be reiterated here. The details of hardware are even well hidden from the systems programmer. Interlan's NI3010 itself provides all Physical layer and most Data Link layer functions.

The NI3010 is a DMA (Direct Memory Access) device and as such must have access to memory in which a packet may be located. The "strapping" of onboard RAM in each SBC, consistent with the design of the multi-user CP/M-86 system, requires that all communications between processors be via common memory. Figure 5.1 illustrates System Configuration of this application.

The program "remote.cmd" is the main routine in this thesis. The same program can be run on each SBC, depending on the number of users desiring remote communications. A slightly modified version of this program is run on the MDS host as "remote5.cmd".

The design is based on a "closed loop" protocol, such that once the SBC will gain the right to access the shared resource, Ethernet, it will not relinquish it until a response message has been received. This "Stop and Wait" protocol makes buffer management almost trivial. The implications however are :

(a) A lost packet (transmit or receive) could deadlock the multi-user system,

(b) The asynchroncus processing capability of a remote host(s) is not utilized (i.e. one heavily loaded host would adversely affect all multiuser CP/M-86 users),

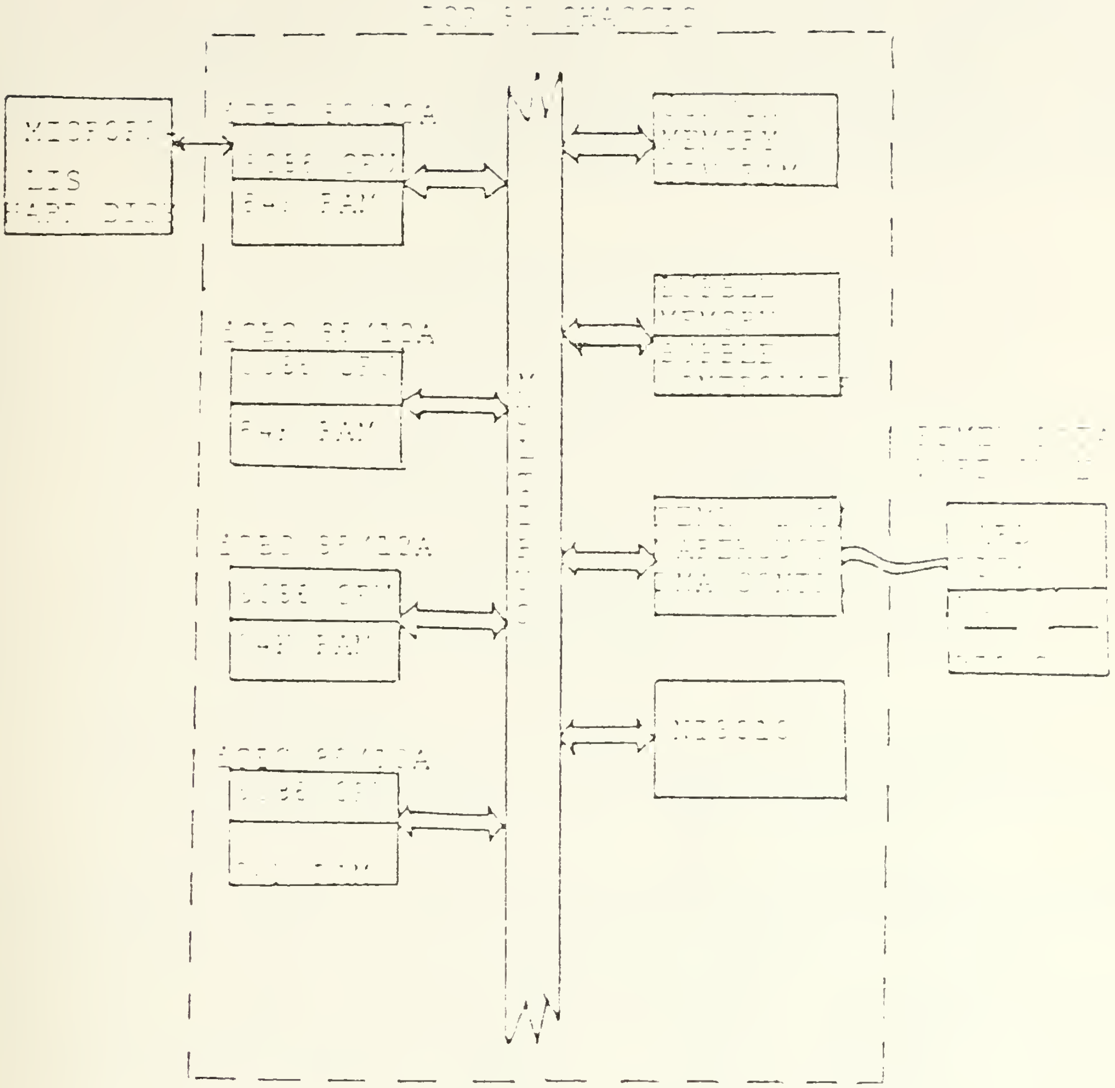

Figure 5.1 System Configuration.

(c) Overall system efficiency is reduced - outbound message packets could be transmitted by each SBC in rapid succession.

In case (a), the high reliability of Ethernet (a 1 bit error in 10 ** 8 - 10 ** 11 bits) precludes the extensive software development necessary to implement an acknowledging Ethernet. In case (b) and (c) it is envisioned that a future version of the communications software will have these considerations as primary objectives. Figure 5.2 illustrates the System perceived by the casual user.

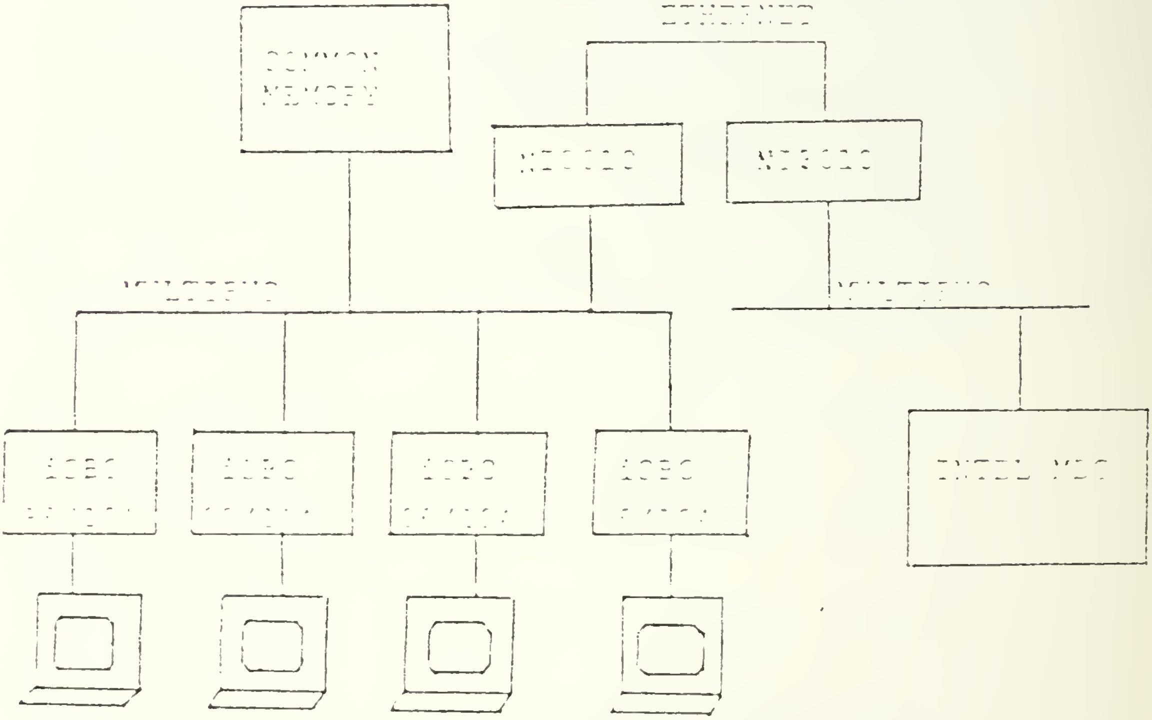

Figure 5.2 User's System view.

Almquist and Steven's Ticket/Server system is used in this implementation. In the Ticket/Server system, when an iSBC 86/12A computer desires to send a message, it requests a ticket number. When its ticket number is one greater than the one previously served, it receives the service. Figure 5.3 illustrates the Common Memory Map Allocation of this

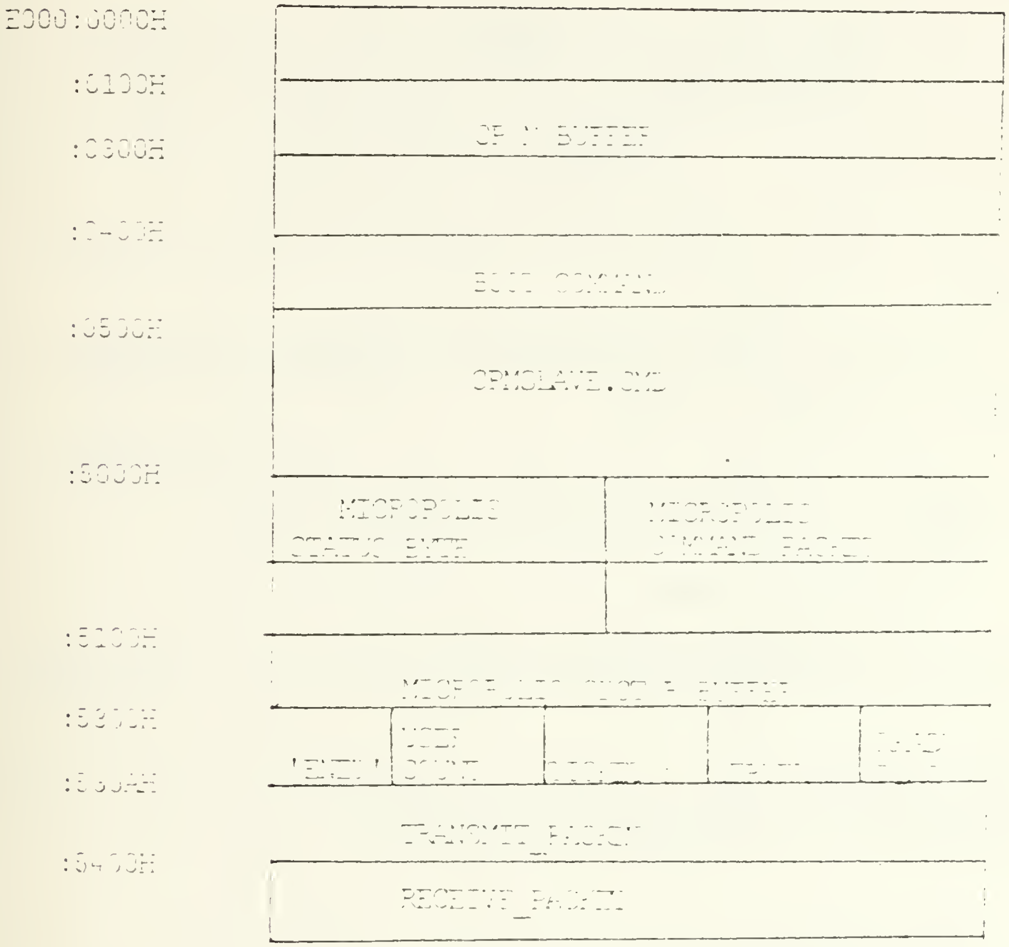

Figure 5.3 Common Memory Allocation Map.

application. For the implementation of this scheme, "ticket" and "server" variables are placed in common memory, since they are shared variables.

2. Ethernet Access

Transmit and Receive sides of the Ethernet access protocol are shown in Figures 5.4 and 5.5.

The Transmit protocol is described as follows:

a. The right to net access must be obtained,

b. The user transfers its packet to common memory,

c. The user initiates a TDD (Transmit_DMA_Done) interrupt,

d. The NI3010 transfers the packet from common memory to its transmit buffer,

e. The NI3010 interrupts the user informing him that the packet is in the NI3010's transmit buffer,

f. The user issues a Load and Send command,

g. The NI3010 transmits the packet via Ethernet.

Figure 5.5 illustrates ETHERNET Access(Receive). The receive protocol is described as follows:

a. The NI3010 issues an interrupt (from the RBA (Receive_Block_Available) mode) to the user when a packet has been received via Ethernet,

b. User initiates a Receive_DMA_Done (RDD) interrupt

c. The packet received is transferred to common memory by NI3010,

d. The NI3010 issues an interrupt,

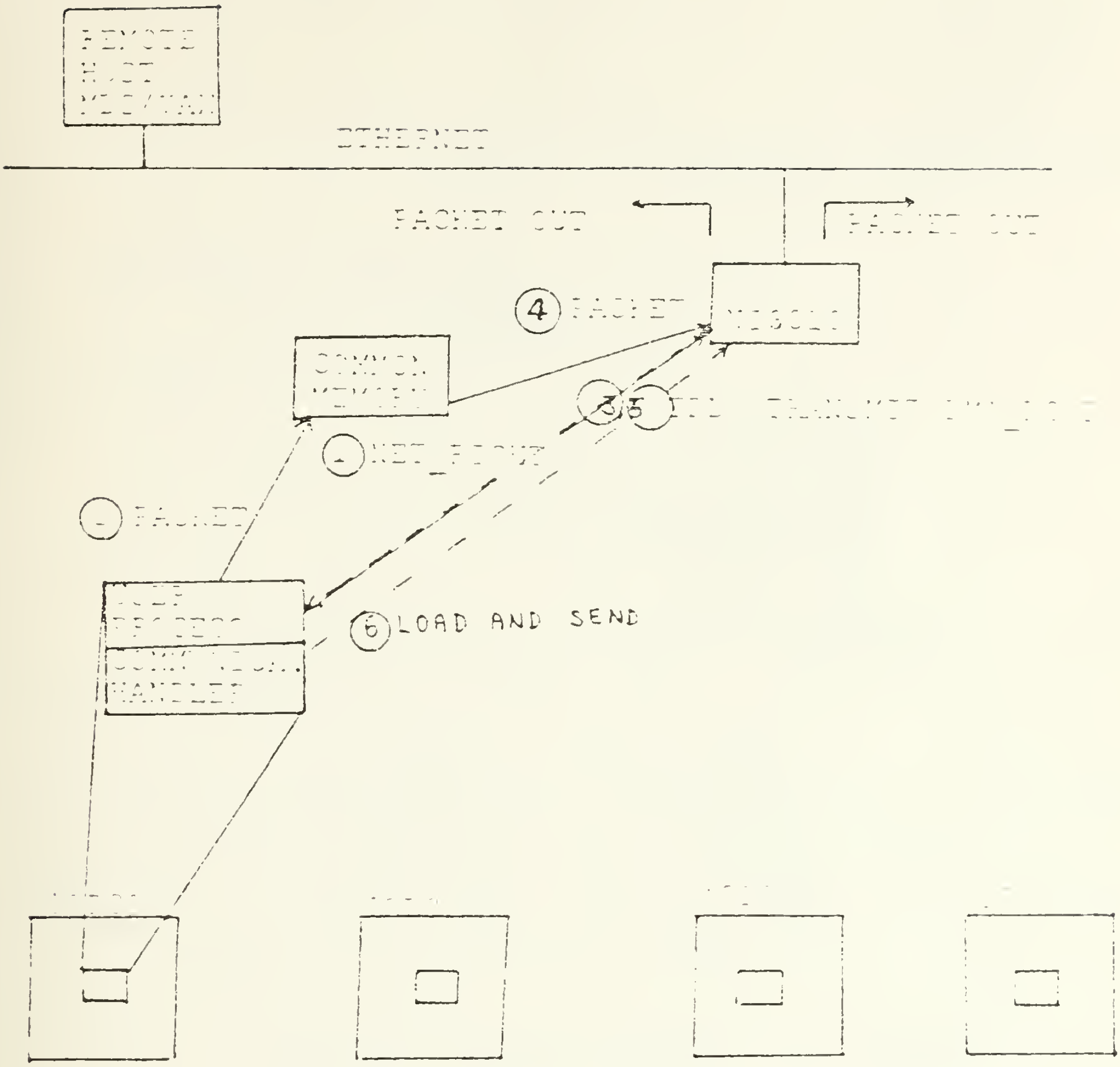

Figure 5.4 ETHERNET Access (Transmit).

e. The user responds to the NI3010 EDI interrupt and transfers the packet from common memory to local memory and processes it.

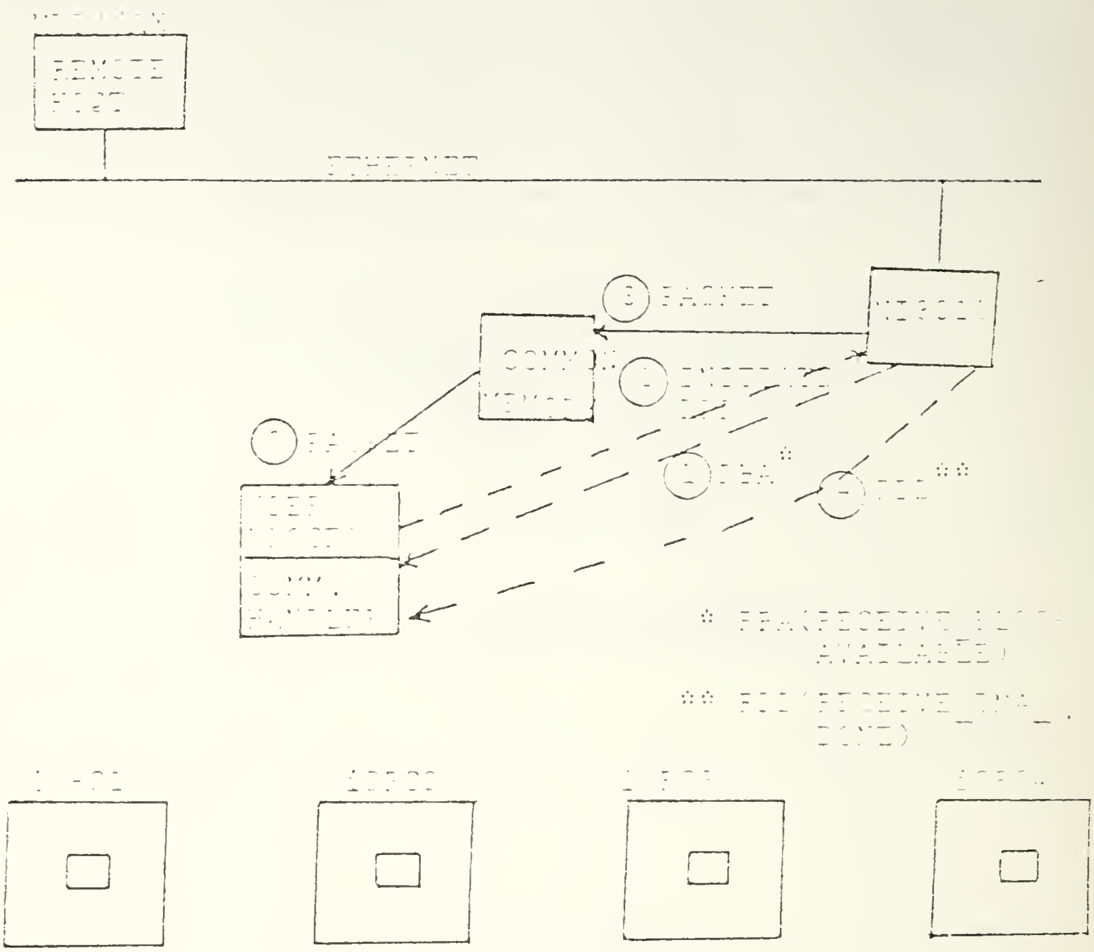

Figure 5.5 ETHERNET Access (Receive).

3. User Dialogue

After loading CP/M and booting the other iSBC's from the terminal 1 as described by Perry [Ref. 1].

The user invokes "remote" .

Program "remote.cmd" (resilent on every logical disk) may be invoked by any user desiring to access a remote host (MDS) via Ethernet.

The applications program responds with :

" REMOTE> "

The users issue the response :

" message " (without the quotation marks)

The applications program prompts with

" destination : "

Answer with " mds " .

The application program prompts with : 'message: ' .

The user then keys in the desired message followed by <CR/IF>.

As soon as the user completes the message (<CR/IF>), the user receives a ticket number, which determines the user's turn to access the Ethernet resource that is multiplexed among all users. This is transparent to the user - the illusion is that he thinks he is the only user on the Ethernet. The code "call request" is the assembly language routine which loops indefinitely (actually AWAIT) until the service number is equal to its ticket number. When it is equal, it returns to "remote.cmd" (actually, within "remote.pli") and the packet is written into the template in common memory by calling "move_to_cm". Then a packet is sent by calling submodule "transmit_packet". When data is ready in common memory, the "data_ready_flag" is set by the interrupt handler indicating that the remote host has responded and data is moved into the authorized user's local RAM by calling move_to_lm subroutine. The next user can then be serviced after the NI30 10 resource is "released" . The message originator is determined as indicated in type_field_b and an appropriate response is issued. User n transmits message to MDS system, where MDS displays the following message :

" Terminal n sent the following message :

(The contents of the message sent by the user)

"Response issued !!! "

4. The NI3010 Transmit Function

The NI3010 transmit function is accomplished in the following manner:

a. The host loads a block of memory in the particular format for each frame to be transmitted.

b. The host loads the three NI3010 address registers with the first address of the host memory block.

c. The host then loads the two NI3010 byte count registers with the number of bytes in the data block.

d. The host then enables a Transmit DMA Done (TDD) interrupt by writing a value of 5 Hex into the Interrupt Enable Register.

e. The NI3010 interrupts the host once the memory block has been transferred into the NI3010 transmit buffer.

f. The host then enables a Receive_Block_Available (RBA) interrupt by loading the Interrupt Enable Register with a value of 4 Hex. This step allows any pending received frames to be handled.

g. The host then commands the NI3010 to send the frame by writing a value of 29 Hex into the Command Register and subsequently reading the Command Status Register.

5. The NI3010 Receive Function

a. The host enables an RBA interrupt.

b. The NI3010, upon receiving a frame, interrupts the host to notify it of frame receipt.

c. The host then writes a value of 0 Hex into the Interrupt Enable Register to disable any other NI3010 interrupts.

d. The host writes values into the three NI3010 address registers to inform the NI3010 where, in host memory , to transfer the data.

e. The host then loads the two NI3010 byte count registers with the hosts buffer size (normally maximum packet size plus a 4 byte header -1522 bytes) .

f. The host then enables the DMA transfer of the data by writing a value of 7 Hex into the Interrupt Enable Register.

g. The NI3010 then interrupts the host upon completion of the transfer. The particular receive data format is used as shown in Figure 3.3.

These steps are repeated for each received frame.

VI. RESULTS AND CONCLUSIONS

A. EVALUATION

A test program "tstether.cmd" is run on the system to show the sequence of users in a random sequence (brought up in the order of tester's choice). Since a closed loop system is used, this random sequence is repeated in the same order as established at that time when the code "remote" is invoked at each terminal. This test program prints an "A" character (at each terminal) after receiving 50 packets, which amounts to 100 round-trip packets.

In 191 seconds 6,144,000 bytes are sent, which equates to 32,167 Bytes/ sec. That is equal to 257,340 bits/sec. Thus, the data rate achieved, during this test was approximately 257 Kbits/sec. This test was performed while 3 terminals attempted to send their messages simultaneously. When only two terminals sent their messages, this only increased the performance by 1 Kbits/sec. The result was 258,015 bits/sec. When only one terminal sent its message this figure was 282,482 bits/sec. Table 1 illustrates these performance metrics.

B. GENERAL CONCLUSION

The principal goal of this thesis has been met. This thesis demonstrates the viability of data communications between iSBC 86/12A's and Intel MDS host system via ETHERNET Local Area Network.

TABLE I

System Performance Measurements

SIMULTANEOUS MESSAGE TRANSMISSION	MAXIMUM DATA RATE(Kbps)
3	257
2	258
1	282

The ICS-80 Industrial Chassis, used in support of this thesis, allows a total of only 12 circuit board slots. The MULTIBUS master slots are odd-numbered and the slave positions are even numbered. Master boards are capable of acquiring and controlling the MULTIBUS.

In this applicaticn, excessive bus master requirements precluded the testing of more than 3 terminals concurrently. This backplane capacity limitation can be alleviated by obtaining an expanded chassis for future implementaticns.

C. FUTURE CONSIDERATIONS

The remote host in this thesis is the CP/M-86 based MDS system. The next thesis can be devoted to the data communications software development between VAX 11/780 and Multiple iSBC's. File transfer, other than message transfer, should alsc be addressed in succeeding research efforts.

APPENDIX A

THE LISTING OF "REMOTE.PLI' MAIN MODULE

```
/***********************************************************************
/*                                                                   */
/* This program, after being linked to multmods,                     */
/* cmaccess, interrpt, and sync Assembler 86 files ,                 */
/* provides the essence of data communication software               */
/* between multi-microcomputers and the host computer.               */
/* The same program is run on each SBC´s . A slightly                */
/* different version is run on MDS side in order to                  */
/* establish the data communication .                                */
/* The iSBC 86/12A´s are used as micros and Intel MDS is             */
/* used as host computer. Multiplexing and Demultiplexing            */
/* is achieved between SBC´s and the MDS system.                     */
/*                                                                   */
/* The program invokes 5 segments :                                  */
/*     1. initialize_pic                                             */
/*     2. perform_command                                            */
/*     3. transmit_packet                                            */
/*     4. HL_interrupt_handler                                       */
/*     5. user_process                                               */
/*                                                                   */
/**********************************************************************/


remote:
   procedure options (main);


   /*    Date:                      30 May 1984


         Programmer:                Izzet Percinler

         Module   Function:         This  module  is  designed  to
                                    function as the cornerstone of
                                    the multiplexing software.
                                    This   software  (after  being
                                    linked    with    ASM    files
                                    multmods, cmaccess, interrpt,
                                    and  sync ) can  be run on any
                                    SBC   and   provide   ETHERNET
                                    service, without any
                                    modificaton.  */
```

```
                   DECLARE

/* The host transfers Ethernet data to the NI3Ø1Ø by getting
up a transmit   block in its own memory. This transmit block
must be in the following   particular format as it is  shown
in Interlan's NI3Ø1Ø manual. (P.18) */



                     1 transmit_data_block static,

  /* The destination address is always 48 bits long.  These
bits  specify  the address of the station(s) for  which  the
frame is intended.  The NI3Ø1Ø requires that all frames have
a destination address  .  The A,B,C bytes of Ethernet address
have been assigned by Xerox.  Interlan has assigned bytes D,
E,  and  F .  The least significant bit      of each byte is
transmitted first. */


                         2 destinatior_address_a
                              bit (8) initial ('Ø2'b4),
                         2 destination_address_b
                              bit (8)initial ('Ø7'b4),
                         2 destination_address_c
                              bit (8)initial ('Ø1'b4),
                         2 destination_address_d
                              bit (8)initial ('ØØ'b4),
                         2 destination_address_e
                              bit (8) ,
                         2 destination_address_f
                              bit (8) ,



                         2 type_field_a      /* packet_type */
                              bit (8) ,
                         2 type_field_b      /* originator */
                              bit (8),


  /*  Minimum data size is 46 bytes,  Maximum data size  is
15ØØ bytes.  In this case data is taken as 238 bytes.  Data
size  may  be  redefined  by changing  data  size  from  238
characters to the other acceptable Ethernet data sizes. */


                         2 data  char (238) varying ,
   /*  The data received by the NI3Ø1Ø (network traffic) is
'DMA'ed' to Common Memory and is in the following particular
format as it is shown in Interlan's NI3Ø1Ø manual .  P.22)*/


                 1 receive_data_block,

                      2 frame_status               bit (8) ,
                      2 null_byte                  bit (8) ,
                      2 frame_length_lsb           bit (8) ,
                      2 frame_length_msb           bit (8) .

   /* The above four bytes of header is added by NI3Ø1Ø when
transferring frame to the host. */

                      2 destination_address_a   bit (8) ,
                      2 destination_address_b   bit (8) ,
                      2 destination_address_c   bit (8) ,
                      2 destination_address_d   bit (8) ,
                      2 destination_address_e   bit (8) ,
                      2 destination_address_f   bit (8) ,


   /* The source address is always 48 bits long.  These bits
contain  the  physical address of the station that sent  the
frame  .  When  transmitting a frame on  the  Ethernet,  the
NI3Ø1Ø automatically inserts the source address. */


                      2 source_address_a           bit (8) ,
                      2 source_address_b           bit (8) ,
                      2 source_address_c           bit (8) ,
                      2 source_address_d           bit (8) ,
                      2 source_address_e           bit (8) ,
                      2 source_address_f           bit (8) ,
                      2 type_field_a               bit (8) ,
                      2 type_field_b               bit (8) ,

   /*  Packets  are fixed length.  256  bytes.  (for  design
simplicity ) 239 bytes of a packet is data field of which  3
bytes are not available to the user : The first is number of
characters  which  contains the length of the  message.  The
others are <CR> and <LF> , which are appended to the varying
character  variable  after the user terminates  his  message
with <CR> <LF>.


                      2 data (239)                 char (1) ,
                    2 crc_msb                  bit (8),
                    2 crc_upper_middle_byte    bit (8),
                    2 crc_lower_middle_byte    bit (8),
                    2 crc_lsb                  bit (8),


                tvar char (238) varying,
                terminal_service bit (8),
                packet_type bit (8) ,
                copy_ie_register bit (8) ,
                (i,j,k) fixed bin (15),
                reg_value bit (8) ,
          border (80) char (1) static initial ((80)'-'),
                enet_init char (4),
                user_count bit (8),
                write_io_port entry (bit (8), bit (8)),
                read_io_port  entry (bit (8), bit (8)),
                move_to_lm entry (bit(16),pointer,
                                       fixed bin (15)),
                move_to_cm entry (pointer, bit(16),
                                       fixed bin (15)),
                initialize_cpu_interrupts      entry,
                enable_cpu_interrupts          entry,
                disable_cpu_interrupts         entry,
                clear_ready_flag               entry,
                initsync                       entry,
                set_ready_flag                 entry,
                increment_user_count           entry,
                decrement_user_count           entry,
                write_bar entry (bit(16));

                /*   end module listing   */



   /*  The 8259A Programmable Interrupt Controller (PIC) is
used in real-time interrupt driven microcomputer systems to
manage  eight  level  interrupts.  It  is  limited  in  this
implementation to respond to interrupts 5 and 6. Interrupt 5
is  used  by the NI3010 and Interrupt 6 is used by  SBC1  to
handle  The  Micropolis  hard  disk I/O  (Refer  to  Parry's
thesis).



                %replace

             /*   codes  specific  to  the  Intel   8259A
          Programmable Interrupt Controller (PIC)       */
```

```
                         icw1_port_address  by 'c0'b4,
       /*  note that */  icw2_port_address  by 'c2'b4,
       /*  icw2,icw4,*/  icw4_port_address  by 'c2'b4,
       /*  and  ocw  */  ocw_port_address   by 'c2'b4,
       /*  use same  */
       /*  port addr */


/* Initialization Command Words (ICWs) are used to set up
   the 8259A  in  an initial  state  of  operation.  ICWs
   are  issued from the processor in a sequential format.

   Operation  Control Words (OCWs) are command words that
   are  sent  to  the  8259A PIC  for  various  forms  of
   operation  ,  such  as  interrupt  masking,  end  of
   interrupt,  priority rotation,  interrupt status.  OCWs
   are  issued  as  needed  to  vary  and  control
   8259A operation.                                    */



                         icw1 by '13'b4,

           /*  single PIC configuration, edge
               triggered input                    */

                         icw2 by '40'b4,

           /*  most significant bits of vectoring
               byte; for an interrupt 5,
               the effective address will be
               (icw2 + interrupt #) * 4 which
               will be (40  hex  +  5)  *  4  =
               114 hex               */

   /*  ICW1 and ICW2 are the minimum amount  of  programming
needed  for  any  type  of  8259A  operation.  The  majority
of  bits  within  these  two ICWs  are  used  to  designate
the interrupt vector starting address. */


                         icw4 by '0f'b4,

   /*  automatic end of interrupt
       and buffered mode/master    */
   /*  OCW1 is used solely for 8259A masking operations.  It
provides  direct  link  to  the  Interrupt  Mask  register
(IMR)  .  The  processor  can  write  to  or  read  from  an
IMR  via  OCW1.  OCW1  sets  and  clears  the  mask  bits  in
the IMR.                                               */


                             ocw1 by '9f'b4.

   /*  unmask  interrupt 5 (bit 5) and mask all others     */

                             /* end 8259a codes */


                             cluster2              by '00'b4,
                             cluster1              by '01'b4,
                             terminal_1            by '01'b4,
                             message_type          by '01'b4,
                       terminal_service_request    by '01'b4,
                             await_enet_access by '02'b4,
                             complete              by '02'b4,
                             in_progress           by '01'b4,
                             mds                   by '05'b4,
                             not_ready             by '00'b4.
   /* ADM-3A specific */     clearscreen           by '^z',
                             addr_rcv_pkt_cm  by '5400'b4,
                             addr_xmit_pkt_cm by '580a'b4,
                             enet_status_addr by '5300'b4,
                             user_count_addr  by '5324'b4;

                 /* include constants specific to the NI3010
                  board                                    */

                    %include 'ni3010.dcl';


/*********************************************************************/


                       /*  Main Body */


          put list (clearscreen);
          put skip;
          put edit ((border (i) do i = 1 to 80)) (a);
          put skip (2) list ('   WELCOME TO THE NET');
          put skip (2);
```

```
      /* Now 8259 is ready to accept interrupt requests *

          call initialize_cpu_interrupts;
          terminal_service = await_enet_access;
    call move_to_lm(enet_status_addr, addr(enet_init), 4);



          if (enet_init ~= 'enet') then
          do;


   /*  First user of ETHERNET for the day,or until the  next
       system crash executes this code */

            user_count = '00'b4;
            enet_init = 'enet';
     call move_to_cm (addr(enet_init),enet_status_addr,4);
     call move_to_cm (addr(user_count), user_count_addr,1);
     call read_io_port (command_status_register,reg_value);
            call perform_command (reset);
     call read_io_port (command_status_register,reg_value);
            call perform_command (go_online);
            call clear_ready_flag;
            call initsync;
          end;
          call increment_user_count;
          copy_ie_register = receive_block_available;
          call user_process;

          call write_io_port(ocw_port_address, 'ff'b4);

/*  Necessary  actually  only  for  iSBC1  as  described  in
Perry's   thesis.  Note:   This  masks  interrupt  5  again_
do  not  want  whatever SBC this code  was  running  on.  to
respond to further NI3010 interrupt signals because user  is
no longer using Ethernet. */


          call decrement_user_count; /* or way out */
          put skip (3);
          put edit ((border (i) do i = 1 to 80)) (a);
          put skip (2);

        /* end main body */


***************************************************************
```

```
/************************************************************/
/*                                                          */
/* This segment initializes Intel 8259A Programmable        */
/* Interrupt Controller (PIC)                               */
/*                                                          */
/************************************************************/



   initialize_pic:
      procedure;

         DECLARE

            write_io_port entry (bit (8) , bit (8));

            call write_io_port (icw1_port_address,icw1);
            call write_io_port (icw2_port_address,icw2);
            call write_io_port (icw4_port_address,icw4);
            call write_io_port (ocw_port_address,ocw1);

   end initialize_pic;


*********************************************************



/**********************************************************
/*                                                       *
/*  This module is used to issue command to the NI3210   *
/*  ETHERNET Communication Controller Board .            */
/*                                                       */
/**********************************************************



   perform_command:
      procedure (command);

         DECLARE

            command bit (8) ,
            reg_value bit (8) ,
            srf bit (8) ,
            write_io_port entry (bit (8) ,
                                 bit (8) ),
            read_io_port  entry (bit (8) ,
         /* end declarations */

            srf = '0'b4;
            call write_io_port (command_register,command);
            do while ((srf & '01'b4) = '00'b4);
               call read_io_port (interrupt_status_reg,
                                              srf);
            end;  /* do while */
            call read_io_port (command_status_register,
                                           reg_value);
      if (reg_value > '01'b4) then
      do;
            /* not (SUCCESS or SUCCESS with Retries) */

          put skip edit ('*** ETHERNET Board Failure ***')
                        (col(35),a);
          stop;
      end; /* itd */

   end perform_command;

************************************************************


/************************************************************/
/*                                                          */
/* This segment will transmit a packet (the structure in    */
/* the declarations section ) -  All fields must be         */
/* appropriately assigned prior to calling this procedure.  */
/*                                                          */
/************************************************************/


   transmit_packet:
      procedure external;


         DECLARE

            srf bit (8) ,
            reg_value bit (8) ,
            write_io_port entry (bit (8) ,
                                 bit (8) ),
            read_io_port entry  (bit (8) ,
                                 bit (8) ),
        enable_cpu_interrupts           entry,
        disable_cpu_interrupts          entry,
    .   write_bar entry (bit(16));




    /* begin */




        call disable_cpu_interrupts;
        do while (copy_ie_register = transmit_dma_done
                                   |
                  copy_ie_register = receive_dma_done);

/*  Present  for future implementations  when  concurrent
    operations   in  progress  - tested  in  LT [illegible]
    Commmmunications software .



         call enable_cpu_interrupts;
         do while (copy_ie_register = transmit_dma_done
                                    |
                   copy_ie_register = receive_dma_done);
         end;
         call disable_cpu_interrupts;

        end;



        copy_ie_register = disable_ni3010_interrupts;
        call write_io_port(interrupt_enable_register,
                           disable_ni3010_interrupts);
        srf = '0'b4;

/* Tell NI3010 where transmit packet may be found */
        call write_bar (addr_xmit_pkt_cm);

        call write_io_port(high_byte_count_reg,'0'b4);
/* 246 bytes */

/* Byte count for NI3010  */
        call write_io_port(low_byte_count_reg,'f6'b4);

        copy_ie_register = transmit_dma_done;
        call write_io_port(interrupt_enable_register,
            call enable_cpu_interrupts;
            do while (copy_ie_register = transmit_ready_reg);
            end;    /* loop until the interrupt handler
                      takes care of the TDD interrupt -
                      it sets IE_REG to 4 */

   /*   Send packet on its way  */
            call write_io_port(command_register,
                               load_and_send);

            do while ((srf & '01'b4) = '00'b4);
               call read_io_port(interrupt_status_reg, srf);
            end;    /* do while */
       call read_io_port(command_status_register,reg_value);

   /*    Prepare for next command -  command_status_register
         MUST  be read after a command is issued,  otherwise
         the host command would be ignored                */




   end transmit_packet;



*************************************************************



/*************************************************************/
/*                                                           */
/* This segment will be active only for the SBC that has     */
/* the ENET access right. All boards will receive NI3010     */
/* interrupts but first conditional check in this handler    */
/* (if terminal_service = in progress )  will cause          */
/* an immediate return from  interrupt for those not         */
/* in service (Only one at a terminal can issue commands     */
/* to the NI3010).                                           */
/*                                                           */
/*************************************************************/


   HL_interrupt_handler:
      procedure external;
```

```
                  /* This routine is called from the low level
                     8086 assembly language interrupt routine.
                     i.e. accessed only via interrupts.   */


         DECLARE

            write_io_port entry (bit (8) ,
                                bit (8) ),
            read_io_port entry (bit (8) ,
                               bit (8) ),
            enable_cpu_interrupts             entry,
            disable_cpu_interrupts            entry,
            write_bar entry (bit(16));

            /*  begin   */



if (terminal_service = in_progress) then
do;
  if (copy_ie_register = receive_block_available) then
     do;
         call write_io_port(interrupt_enable_register,
                                 disable_ni3212_interrupts);
                           call write_bar (addr_rcv_pkt_cr);
        call write_io_port(high_byte_count_reg,'01'b4);
        /* 260 bytes */

         call write_io_port(low_byte_count_reg,'04'b4);

               /* initiate receive DMA */

                   copy_ie_register = receive_dma_done;
          call write_io_port(interrupt_enable_register,
                                            receive_dma_done);

end;    /* do */
else
    if (copy_ie_register = receive_dma_done) then
        do;
            call set_ready_flag;
            /* informs a terminal that data is ready */
            copy_ie_register = receive_block_available;

          call write_io_port(interrupt_enable_register,
                                   receive_block_available);

end;  /* if then do */
```

```
            if  copy_ie_register = transmit_dma_done
            then do;
               copy_ie_register = receive_block_available;

              call write_io_port interrupt_enable_register,
                                     receive_block_available ;
            end;    /* if then do */
     end;

   end HL_interrupt_handler;



**********************************************************************




/*******************************************************************/
/*                                                                 */
/* This procedure prompts the user for input to invoke             */
/* remote communications -  note that the only acceptable          */
/* input following remote> is 'message'  and  following            */
/* Destination: is 'mds' (without the quotes in both               */
/* case.)                                                          */
/*                                                                 */
/*******************************************************************/



   user_process:
      procedure;

         DECLARE

            p pointer,
            convert_to_binary fixed bin(7) based  (p),
            terminal bit (8),
            packet_type bit (8),
            cluster bit (8),
            (i,j,k) fixed bin (15),
            service_response char (10) varying,
            destination_response char (3),
            move_to_lm entry (bit(16),pointer,
                              fixed bin (15)),
            move_to_cm entry (pointer, bit(16),
                              fixed bin (15)),
            data bit (8) static init ('00'b4),
            read_ready_flag entry returns (bit(8)),
            request entry,
        release entry,
        clear_ready_flag entry;


     put skip list ('Terminal number: ');
     get edit (terminal) (b4(2));
     transmit_data_block.type_field_b = terminal;
     service_response = 'continue';
     do while (service_response ^= 'exit');
        data = not_ready;
        put skip list ('REMOTE> ');
        get list (service_response);
        if (service_response = 'message') then
        do;
            packet_type = message_type;
            put skip list ('Destination: ');
            get list (destination_response);
            if (destination_response = 'mds') then
            do;
               cluster = cluster1;
               if (cluster = cluster2) then
               do;
     transmit_data_block.destination_address_e = '15'b4;
     transmit_data_block.destination_address_f = 'ea'b4;
               end;

            else     /* cluster = cluster1 */
            do;
     transmit_data_block.destination_address_e = '04'b4;
     transmit_data_block.destination_address_f = '9a'b4;
            end;  /* else if */
            if (packet_type = message_type) then
     transmit_data_block.type_field_a = message_type;
            put skip list ('Message: ');
            read into (tvar);
            read into (tvar); /* must read twice */
            transmit_data_block.data = tvar;

    /* now must get the right to use the ETHERNET
       resource multiplexed among all users.
       More directly the limited resource is the
       transmit packet template in common memory */

       call request;  /* Intel 8086 assembly language
                         routine - will loop indefinitely
                         until my service number
                         is reached                     */
               terminal_service = in_progress;

/* my turn!!! So, write the transmit packet
               to the template in common memory    */
            call move_to_cm (addr(transmit_data_block),
                                 addr_xmit_pkt_cm,
                                 246);
            call transmit_packet;      /* sent it  */

            do while (data = not_ready);
    data = read_ready_flag();   /*  wait for response */

/* 8086 routine that reads
  flag which interrupt handler (communication handler is
  integrated into this ) sets to one when data is
  available     */


            end;

          /* data is ready in common memory -
            remote host has responded: so get data */

            call move_to_lm  addr_rcv_pkt_cm,
                                addr(receive_data_block),
                                260);

            call clear_ready_flag;
            terminal_service = complete;
            call release;   /* allows next user to be
                                  serviced    */

             /* display response on CRT */
   if (receive_data_block.type_field_a = message_type)
             then do;

       /* get type field_b to determine what host
          is trying to communicate with this terminal
          and then give an appropriate response */

         if (receive_data_block.type_field_b = mds
         then do;
            put skip list ('MDS responds with: ');
            i = 2;

 /*  Easy method to "convert" character data to arithmetic
    data - "overlay" character variable with arithmetic
    variable name.                                     */


            p = addr ( receive_data_block.data 1));
          do while (i <= ( convert_to_binary + 1 ));
               put edit(receive_data_block.data(i))
                          (a(1));
          i = i + 1;
               end;
```

```
                        put skip;
                    end;  /* do */
                 end; /* do */
              end;
           end;
        end;
     end ;  /* user_process */

end;  /* remote */
```

APPENDIX E

THE MODULE LISTING OF "SYNC.A86"

```
;Prog Name      :Sync.A86
;Date           :15 May 1984
;Written by     :Izzet Percirler
;For            :Thesis
;Advisor        :Professor Kodres
;Purpose        :Provide synchronizations of CPM 86
;                users requesting service of the
;                Ethernet.
;
;
;*******************************************************

;                 Synchronization Routine

;*******************************************************

public release
public request
public initsync
public read_ready_flag
public clear_ready_flag
public set_ready_flag
public increment_user_count
public decrement_user_count


;*******************************************************

;                           Equates

;*******************************************************

    common_memory_seg equ 0e000h
         dcount          equ 100     ;bus contention time delay

;*******************************************************

;                         Subroutines

;*******************************************************

        cseg

ticket:                    ;return the next ticket number in
                           ;bx
        push ax
repeat_test:
        xor  ax,ax                      ;set reserved value
        lock xchg ax,next               ;get ticket number
        test ax,ax
        jz   repeat_test                ;repeat if reserved
        mov  bx,ax                      ;return next ticket
        inc  ax
        jnz  tic1
        inc  ax                         ;skip reserved value
tic1:   mov  next,ax                    ;increment ticket number
        pop  ax
        ret

;------------------------------------------------------------------

 await:                         ;wait for server number to match
                                ;the customers ticket number (passed
                                ;in bx. To reduce bus contention, a
                                ;delay is used between periodic
                                ;checks of the server number
        push cx
again:
        cmp  bx,server                  ;if ticket = server
        je   awa2                       ;continue process
        mov  cx,dcount                  ;if not, insert delay
awa1:   dec  cx
        jnz  awa1
        jmp  again                      ;check server again
awa2:   pop  cx
        ret

;------------------------------------------------------------------

 advance:                       ;increment server number to next
                                ;value
        inc  server                     ;server=server+1
        jnz  adv1
        inc  server                     ;skip reserved value
adv1:   ret

;------------------------------------------------------------------

 request:                       ;get a ticket number and wait to be
                                ;served

        push es
        push ax
        push bx
        mov  ax,common_memory_seg  ;set es to address common
        mov  es,ax                 ;memory
        call ticket                     ;get ticket number
        call await                      ;wait to be served
        pop  bx
        pop  ax
        pop  es
        ret

;-----------------------------------------------------------------

 release:                     ;adv server number on completion
                              ;of read or write operation

        push es
        push ax
        mov  ax,common_memory_seg ;set es to address common
        mov  es,ax                ;memory
        call advance              ;inc server number
        pop  ax
        pop  es
        ret

;-----------------------------------------------------------------

 initsync:                 ;initialize sequencer variables

        push es
        push ax
        mov  ax,common_memory_seg  ;set es to address common
        mov  es,ax                 ;memory
        mov  ax,1                  ;server=next=1
        mov  server,ax
        mov  next,ax
        pop  ax
        pop  es
        ret


;-----------------------------------------------------------------


read_ready_flag:

        push es
        push bx
        mov  bx, common_memory_seg ;set es to address common
        mov  es, bx                ;memory
        mov  al, ready_flag
        pop  bx
        pop  es
        ret
```

```
;-------------------------------------------------------------------


clear_ready_flag:


         push es
         push ax
         mov  ax, common_memory_seg ;set es to address common
         mov  es, ax                ;memory
         mov  ready_flag, 0         ;ready_flag = 0
         pop  ax
         pop  es
         ret


;-------------------------------------------------------------------



set_ready_flag:

         push es
         push ax
         mov  ax, common_memory_seg ;set es to address common
         mov  es, ax                ;memory
         mov  ready_flag, 1         ;ready_flag = 1
         pop  ax
         pop  es
         ret


;-------------------------------------------------------------------



increment_user_count:

         push es
         push ax
         mov  ax, common_memory_seg ;set es to address common
         mov  es, ax                ;memory
         add  user_count, 1
         pop  ax
         pop  es
         ret



;-------------------------------------------------------------------
```

```
decrement_user_count:

        push es
        push ax
        mov  ax, common_memory_seg ;set es to address common
        mov  es, ax                ;memory
        sub  user_count, 1
        pop  ax
        pop  es
        ret

;**********************************************************************

;                              Data

;**********************************************************************


        eseg              ;only one set of sequencer variables
                          ;exist in common memory
                          ;accessed via es

        org     5384h

                user_count        rb 1
                server            rw 1
                next              rw 1
                ready_flag        rb 1

        end
```

APPENDIX C

THE MODULE LISTING OF "MULTMODS.A86"

```
;Prog Name      :Multmods.A86
;Date           :17 May 1984
;Written by     :Izzet Percinler
;For            :Thesis
;Advisor        :Professor Kodres
;Purpose        :Loads bus address and byte count registers
;                and writes to specific I/O ports.
;
;

public write_io_port
public read_io_port
public write_bar

;**********************************************************************


write_io_port:

     ; Parameter Passing Specification:

     ;                    entry                    exit
     ;
     ; parameter 1     <port address>          <unchanged>
     ;
     ; parameter 2     <value to be outputted   unchanged
     ;
     ;

          dseg

          port_address   rb   1

          cseg


          push bx! push si! push dx! push ax
          mov  si, [bx]
          mov  al, [si]
          mov  port_address, al
          mov  si, 2[bx]
          mov  al, [si]
          mov  dl, port_address
          mov  dh, 00h
          out  dx, al
```

```
        pop ax! pop dx! pop si! pop bx
        ret

;*****************************************************************************

read_io_port:

    ; Parameter Passing Specification
    ;
    ;                       entry                     exit
    ;
    ; parameter 1     <port address>            <unchanged
    ; parameter 2     <meaningless>           <register val

        cseg
        push bx! push si! push dx! push ax
        mov  si,  [bx]
        mov  al,  [si]
        mov  port_address,  al
        mov  si, 2[bx]
        mov  dl, port_address
        mov  dh,  ØØh
        in   al,  dx
        mov  [si], al
        pop  ax! pop dx! pop si! pop bx!
        ret

;*****************************************************************************

write_bar:

    ; Parameter Passing Specification
    ;
    ; parameter 1 (and only): the address of the data block to
    ;                         transmitted or received.

        dseg

        e_bar_port      equ  Øb9h
        h_bar_port      equ  Øbah
        l_bar_port      equ  Øbbh
        temp_e_byte     rb    1
        temp_es         rw    1

        cseg

        ; This module computes a 24 bit address from a 32 bi
        ; address - actually it's a combination of the ES re
        ; and the IP passed via a parameter list.
```

```
          push bx! push ax! push cx! push es! push dx! push si
          mov  dx,  0e000h                       ; common memory s
          mov  es,  dx
          mov  temp_es, es
          mov  dx,  es
          mov  si,  [bx]
          mov  ax,  [si]
          mov  cl,  12
          shr  dx,  cl
          mov  temp_e_byte,   dl
          mov  dx,  temp_es
          mov  cl,  4
          shl  dx,  cl
          add  ax,  dx
          jnc  no_add
add_1:    inc  temp_e_byte
no_add:   out  l_bar_port, al
          mov  al, ah
          out  h_bar_port, al
          mov  al, temp_e_byte
          out  e_bar_port, al
          pop si! pop dx! pop es! pop cx! pop ax! pop bx
          ret


end
```

APPENDIX I

THE MODULE LISTING OF "CMACCESS.A86"

```
;Prog Name          :cmaccess.a86
;Date               :19 May 1984
;Written by         :Izzet Percinler
;For                :Thesis
;Advisor            :Professor Kodres
;Purpose            :Moves data from local memory to common
;                    memory until data bytes are exhausted.
;

public move_to_cm
public move_to_lm

;*****************************************************************


move_to_cm:

; Module Interface Specification:

;                    entry                        exit

; parameter 1     from (local memory)          unchanged

; parameter 2     to   (common memory)         unchanged

; parameter 3     number_bytes                 unchanged

;  Parameters 1 and 2 are offsets only

        push    es
        push    cx
        push    ax
        push    bx
        push    si
        push    di

        mov     si,   [bx]
        mov     si,   [si]   ; si contains parameter 1
        mov     di,   2[bx]
        mov     di,   [di]   ; di contains parameter 2
        mov     bx,   4[bx]
```

```
          mov     cx,   [bx]  ;cx contains the number of bytes
          mov     ax,   0e000h
          mov     es,   ax
rep       movsb                 ; move data until cx = 0
          pop     di
          pop     si
          pop     bx
          pop     ax
          pop     cx
          pop     es
          ret

move_to_lm:

; Module Interface Specification:

;                  entry                     exit

; parameter 1     from (common memory)      unchanged

; parameter 2     to    (local memory)     unchanged

; parameter 3     number_bytes              unchanged

;  Parameters 1 and 2 are offsets only


          push    ds
          push    cx
          push    ax
          push    bx
          push    si
          push    di

          mov     si,   [bx]
          mov     si,   [si]  ; si contains parameter 1  (from)
          mov     di,   2[bx]
          mov     di,   [di]  ; di contains parameter 2  (to)
          mov     bx,   4[bx]
          mov     cx,   [bx]  ;cx contains the number of bytes
          mov     ax,   0e000h
          mov     ds,   ax
rep       movsb                 ; move data until cx = 0
          pop     di
          pop     si
          pop     bx
          pop     ax
          pop     cx
          pop     ds
          ret
```

APPENDIX E

THE MODULE LISTING OF "INTERRPT.A86

```
;Prog Name              :Interrpt.a86
;Date                   :19 May 1984
;Written by             :Izzet Percinler
;For                    :Thesis
;Advisor                :Professor Kodres
;Purpose                :Provides the initialization of CPU
;                        interrupts, and it enables and
;                        disables CPU interrupts.
;
;


public initialize_cpu_interrupts
public enable_cpu_interrupts
public disable_cpu_interrupts
extrn hl_interrupt_handler : far


;-----------------------------------------------------


initialize_cpu_interrupts:


    ; Module Interface Specification:

    ;      Caller:          Ethertest(PL/I) Procedure

    ;      Parameters:      NONE

    initmodule cseg common
               org 114h
               int5_offset    rw 1
               int5_segment   rw 1

               cseg
               push bx
               push ax
               mov  bx,  offset interrupt_handler
               mov  ax,  0
               push ds
               mov  ds,  ax
               mov  ds:int5_offset, bx
               mov  bx, cs
               mov  ds:int5_segment, bx
               pop  ds
```

```
                pop  ax
                pop  bx
                sti
                ret

;------------------------------------------------------------

enable_cpu_interrupts:

     ; Module Interface Specification:

     ;      Caller:          Ethertest(PL/I) Procedure

     ;      Parameters:      NONE

                sti
                ret

;------------------------------------------------------------

disable_cpu_interrupts:

     ; Module Interface Specification:

     ;      Caller:          Ethertest(PL/I) Procedure
     ;      Parameters:      none

                cli
                ret

;------------------------------------------------------------

interrupt_handler:


                ; IP, CS, and flags are already on stack
                ; save all other registers

                push ax
                push bx
                push cx
                push dx
                push si
                push di
                push bp
                push ds
                push es
      call hl_interrupt_handler ;high level source routine
                                     ;In Ethertest module PL/I.
                ; restore registers

                pop es
                pop ds
                pop bp
                pop di
                pop si
                pop dx
                pop cx
                pop bx
                pop ax
                sti
                iret

  end
```

APPENDIX E

THE MODULE LISTING OF 'NI3Ø1Ø.DCL'

```
%replace

  /*                    I/O port addresses

        These values are specific to the use of the INTERLAN
    NI3Ø1Ø MULTIBUS to ETHERNET interface board. Any change
    to the I/O port address of 'ØØbØ' hex (done so with a
    switch)  will  require  a change to  these  addresses to
    reflect that change.


            command_register                    by 'bØ'b4,
            command_status_register             by 'b1'b4,
            transmit_data_register              by 'b2'b4,
            interrupt_status_reg                by 'b5'b4,
            interrupt_enable_register           by 'b8'b4,
            high_byte_count_reg                 by 'bc'b4,
            low_byte_count_reg                  by 'bd'b4,


     /*  end of I/O port addresses */

     /*  Interrupt enable status register values */
            disable_ni3Ø1Ø_interrupts           by 'ØØ'b4,
            ni3Ø1Ø_intrpts_disabled             by 'Ø7'b4,
            receive_block_available             by 'Ø4'b4,
            transmit_dma_done                   by 'Ø6'b4,
            receive_dma_done                    by 'Ø7'b4,


     /*   end register values */

     /*   Command Function Codes */

            module_interface_loopback           by 'Ø1'b4,
            internal_loopback                   by 'Ø2'b4,
            clear_loopback                      by 'Ø3'b4,
            go_offline                          by 'Ø8'b4,
            go_online                           by 'Ø9'b4,
            onboard_diagnostic                  by 'Øa'b4,
            load_transmit_data                  by '28'b4,
            load_and_send                       by '29'b4,
            reset                               by '3f'b4;

     /*   end Command Function Codes         */
```

APPENDIX G

THE MODULE LISTING OF 'REMOTE5.PLI'

```
remote5:          procedure options (main) ;


   /*   Date                :  15 May 1984

        Programmer          :  Izzet Percinler

        Module Function :  This module is designed to  send
                           and receive packets via ETHERNET
                           to act as a distant host to the
                           the multi-user CP/M system.
                                                          */




                DECLARE

                    1    transmit_data_block static,

                         2 destination_address_a
 /*                ----\*/     bit (8) initial ('02'b4),
 /*  assigned      |    */   2 destination_address_b
 /*     by         ---->*/     bit (8)initial ('07'b4),
 /*   XEROX        |    */   2 destination_address_c
 /*                ---->*/     bit (8)initial ('01'b4),
                             2 destination_address_d
 /*                ----\*/     bit (8)initial ('00'b4),
 /*  assigned      |    */   2 destination_address_e
 /*     by         ---->*/     bit (8) , /* must be assigned */
 /*  INTERLAN      |    */   2 destination_address_f
 /*                ---->*/     bit (8) , /* must be assigned */
                             2 type_field_a
                               bit (8) , /* must be assigned */
                             2 type_field_b
                               bit (8)  initial ('00'b4),
                             2 data
                               char (238) varying,


                    1 receive_data_block,

                         2 frame_status              bit (8).
                         2 null_byte                 bit (8) .
                         2 frame_length_lsb          bit (8) .
```

```
                 2 frame_length_msb         bit (8) ,
                 2 destination_address_a   bit (8) ,
                 2 destination_address_b   bit (8) ,
                 2 destination_address_c   bit (8) ,
                 2 destination_address_d   bit (8) ,
                 2 destination_address_e   bit (8) ,
                 2 destination_address_f   bit (8) ,
                 2 source_address_a        bit (8) ,
                 2 source_address_b        bit (8) ,
                 2 source_address_c        bit (8) ,
                 2 source_address_d        bit (8) ,
                 2 source_address_e        bit (8) ,
                 2 source_address_f        bit (8) ,
                 2 type_field_a            bit (8) ,
                 2 type_field_b            bit (8) ,
                 2 data (238)              char (1) ,
                 2 crc_msb                 bit (8) ,
                 2 crc_upper_middle_byte   bit (8) ,
                 2 crc_lower_middle_byte   bit (8) ,
                 2 crc_lsb                 bit (8) ,


              originator bit (8),
              rcvd_packet_type bit (8) ,
              copy_ie_register bit (8) ,
              (i,j,k) fixed bin (15),
              reg_value bit (8) ,
              operation bit (8) ,
      border (80) char (1) static initial ((80)'-'),

              /*   Modules external to this module */

              write_io_port entry (bit (8) ,bit (8)) ,
              read_io_port  entry (bit (8) ,bit (8)) ,
              initialize_cpu_interrupts       entry,
              enable_cpu_interrupts           entry,
              disable_cpu_interrupts          entry,
              write_bar entry (pointer);

              /*   end module listing  */






              %replace
   /*  codes specific to the Intel 8259a Programmable
```

```
                         Interrupt Controller (PIC)      */

                         icw1_port_address         by 'c0'b4,
/*  note  that */        icw2_port_address         by 'c2'b4,
/*  icw2,icw4, */        icw4_port_address         by 'c2'b4,
/*  and ocw    */        ocw_port_address          by 'c2'b4,
          /* use same  */
          /* port addr */

                              /* note: icw ==  initialization
                                               control
                                               word

                                        ocw ==> operational
                                                command
                                                word        */


                                icw1                       by '13'b4,
                              /* single PIC configuration, edge-
                                triggered input                  */

                                icw2                       by '40'b4,

                    /* most significant bits of vectoring
                                     byte; for an interrupt 5,
                                the effective address will be
                               (icw2 + interrupt #) * 4 which
                                   will be (40 hex + 5) * 4 =
                                        114 hex               */

                         icw4                              by '0f'b4,

                              /* automatic end of interrupt
                              and buffered mode/master      */

                         ocw1                              by 'df'b4,

                           /* unmask interrupt 5 (bit 5) and
                              mask all others               */

                                /* end 8259a codes */

                         clearscreen                       by ' z',
                         cluster0                          by '00'b4,
                         cluster1                          by '01'b4,
                         mds                               by '05'b4,
                         await_packet                      by '00'b4,
                         packet_received                   by '01'b4,
                         message_type                      by '01'b4;

                   /* include constants specific to the NI3010
```

```
                board                                                       */

                  %include 'ni3210.dcl';

/***************************************************************************/

                    /*  Main Body */


          put list (clearscreen);
          put skip;
          put edit ((border (i) do i = 1 to 80)) (a);
          operation = await_packet;
      call  read_io_port (command_status_register,reg_val);
          call initialize_pic;
          call initialize_cpu_interrupts;
          call perform_command (go_online);
          copy_ie_register = receive_block_available;
          call system_process;
          call perform_command (reset);
          put skip (3);
          put edit ((border (i) do i = 1 to 80)) (a);
          put skip (2);

        /* end main body */

/***************************************************************************/



            initialize_pic:  procedure;

                DECLARE

                    write_io_port entry (bit (8), bit (8));

              call write_io_port (icw1_port_address,icw1);
              call write_io_port (icw2_port_address,icw2);
              call write_io_port (icw4_port_address,icw4);
              call write_io_port (ocw_port_address,ocw1);

              end initialize_pic;

/***************************************************************************/


              perform_command:         procedure (command);

                  DECLARE
                      command bit (8) ,
                       reg_value bit (8) ,
                       srf bit (8) ,
                       write_io_port entry (bit (8) ,
                                            bit (8) ),
                       read_io_port  entry  bit (8) ,
                                            bit (8) );

                       /* end declarations */

                       srf = '0'b4;
              call write_io_port (command_register,command);
                do while ((srf & '01'b4) = '00'b4);
                    call read_io_port (interrupt_status_reg,
                                                          srf);
                end;  /* do while */
                call read_io_port (command_status_register,
                                   reg_value);
                if (reg_value > '01'b4) then
                do;
                /* not (SUCCESS or SUCCESS with retries) */

           put skip edit ('*** ETHERNET Board Failure ***')
                                                 (col(20),a);
                              stop;
                       end; /* iti */


                end perform_command;




/*****************************************************


           transmit_packet: procedure (cluster,packet_type)
                            external;



                    DECLARE
                            cluster bit (8) ,
                            packet_type bit (8) ,
                            srf bit (8) ,
                            reg_value bit (8) ,
                            write_io_port entry (bit (8) ,
                                                 bit (8) ),
                            read_io_port entry  (bit (8) ,
                                                 bit (8) ),
                       enable_cpu_interrupts         entry,
                       disable_cpu_interrupts        entry,
                    write_bar entry  pointer ;







            /* begin */
            call disable_cpu_interrupts;
       do while (copy_ie_register = transmit_dma_done
                                    !
              copy_ie_register = receive_dma_done ;

            call enable_cpu_interrupts;
       do while (copy_ie_register = transmit_dma_done
                                    !
              copy_ie_register = receive_dma_done ;
              end;
              call disable_cpu_interrupts;
            end;
         copy_ie_register = disable_ni3010_interrupts;
         call write_io_port(interrupt_enable_register,
                             disable_ni3010_interrupts ;
            srf = '0'b4;
            if (cluster = cluster2) then
            do;
      transmit_data_block.destination_address_e = '[illegible]'b4;
      transmit_data_block.destination_address_f = '[illegible]'b4;
            end;

            else    /* cluster = cluster1 */
            do;
      transmit_data_block.destination_address_e = '[illegible]'b4;
      transmit_data_block.destination_address_f = '[illegible]'b4;
            end;  /* else do */
            if (packet_type = message_type) then
       transmit_data_block.type_field_a = message_type;
          call write_bar (addr(transmit_data_block));
call write_io_port(high_byte_count_reg,'0'b4);/*246 bytes*/
        call write_io_port(low_byte_count_reg,'f6'b4);
            copy_ie_register = transmit_dma_done;
         call write_io_port(interrupt_enable_register,
                                    transmit_dma_done);
            call enable_cpu_interrupts;
       do while (copy_ie_register = transmit_dma_done);
          end;   /* loop until the interrupt handler
                      takes care of the TDD interrupt -
                        it sets IE_REG to 4 */
            call write_io_port(command_register,
                                    load_and_send);
            do while ((srf & '01'b4) = '00'b4);
         call read_io_port(interrupt_status_reg, srf);
```

```
                    end;    /* do while */
        call read_io_port(command_status_register,new_value);

                 end transmit_packet;

/********************************************************************************


                 HL_interrupt_handler: procedure external;


                   /* This routine is called from the low level
                    8086 assembly language interrupt routine */

                      DECLARE

                             write_io_port entry (bit(8),
                                                  bit(8)),
                             read_io_port entry (bit(8),
                                                 bit(8)),
                         enable_cpu_interrupts           entry,
                         disable_cpu_interrupts          entry,
                             write_bar entry (pointer);

                      /*  begin   */
                      call disable_cpu_interrupts;
                 call write_io_port(interrupt_enable_register,
                                     disable_all_interrupts);
               if (copy_ie_register = receive_block_available)
                      then do;
                    call write_bar (addr(receive_data_buffer));

/* 262 bytes */
               call write_io_port(high_byte_count_reg,'01'h);
               call write_io_port(low_byte_count_reg,'06'h);

                          /* initiate receive DMA */

                         copy_ie_register = receive_dma_done;
                 call write_io_port(interrupt_enable_register,
                                            receive_dma_done);

                      end;    /* do */
                      else
                 if (copy_ie_register = receive_dma_done) then
                          do;
                              operation = packet_received;
                copy_ie_register = receive_block_available;
                call write_io_port(interrupt_enable_register,
                                  receive_block_available);
                          end;  /* if then do */
                    else
                  if (copy_ie_register = transmit_dma_done
                       then do;
              copy_ie_register = receive_block_available;
              call write_io_port(interrupt_enable_register,
                                          receive_block_available);

                         end;    /* if then do */
               end FL_interrupt_handler;

/*************************************************************************/




        system_process: procedure;

                DECLARE

                        p pointer,
                convert_to_binary fixed bin (7) based (p),
                        tvar char (40) varying,
                        i fixed bin (15),
                write_io_port entry (bit (8),bit (8));

                copy_ie_register = receive_block_available;
             call write_io_port (interrupt_enable_register,
                                          receive_block_available);
      put skip list ('AWAITING NETWORK COMMUNICATIONS...');
                put skip;
                do while ('1'b);
                  if (operation = packet_received) then
                        do;
                            call disable_cpu_interrupts;
        if (receive_data_block.type_field_a = message_type)
                            then do;
                           rcvd_packet_type = message_type;
              originator = receive_data_block.type_field_b;

                 put skip(3) edit ('Terminal ', originator,
                       ' sent the following message: ')
                                               (a,b4(2),a);
                                 i = 2;
                     p = addr(receive_data_block.data(1));
                   do while (i <= (convert_to_binary + 1));
               put edit (receive_data_block.data(i))(a(1));
                                   i = i + 1;
                                end;



                    transmit_data_block.type_field_b = mds;
```

```
if (originator = '01'b4 | originator = '02'b4
 | originator = '03'b4 | originator = '04'b4)
                         then
                         do;

                      if   (originator = '01'b4) then
           tvar = 'Terminal 1 message was received!';
                      else
                         if (originator = '02'b4) then
           tvar = 'Terminal 2 message was received!';
                              else
                         if (originator = '03'b4) then
           tvar = 'Terminal 3 message was received!';
                              else
                         if (originator = '04'b4) then
           tvar = 'Terminal 4 message was received!';

                      transmit_data_block.data = tvar;

                         end;
                         operation = await_packet;
                    end;
                    call enable_cpu_interrupts;
           put skip(2) list ('Response ISSUED !!!');
  call transmit_packet (cluster2, rcvd_packet_type);
                 end;
          end;  /* do while forever */

    end system_process;

/*****************************************************************/


end; /* procedure remote5 */
```

APPENDIX H

THE MODULE LISTING OF 'R5MOD.PLI'

```
r5mod:   procedure options (main);

   /*    Date:                    15 May 1984

         Programmer:              Izzet Percirler

         Module  Function:  This module is designed to send
                            and receive packets via [illegible]
                            to act as a distant station [illegible]
                            the multi-user CP/M system.



               DECLARE

                       1    transmit_data_block static,

                              2 destination_address_a
 /*                 ---->*/        bit (8) initial ('02'b4),
 /*  assigned     |     */    2 destination_address_b
 /*     by        ---->*/         bit (8)initial ('07'b4),
 /*   XEROX       |     */    2 destination_address_c
 /*               ---->*/         bit (8)initial ('01'b4),
                              2 destination_address_d
 /*               ---->*/         bit (8)initial ('30'b4),
 /*  assigned     |     */    2 destination_address_e
 /*     by        ---->*/       bit (8) , /* must be assigned [illegible]
 /*  INTERLAN     |     */    2 destination_address_f
 /*               ---->*/       bit (8) , /* must be assigned [illegible]
                              2 type_field_a
                                bit (8) , /* must be assigned [illegible]
                              2 type_field_b
                                  bit (8)  initial ('00'b4),
                              2 data
                                  char (238) varying,


                      1 receive_data_block,

                              2 frame_status              bit (8) ,
                              2 null_byte                 bit (8) ,
                    2 frame_length_lsb             bit (8) ,
                    2 frame_length_msb             bit (8) ,
                    2 destination_address_a        bit (8) ,
                    2 destination_address_b        bit (8) ,

                    2 destination_address_c        bit (8) ,
                    2 destination_address_d        bit (8) ,
                    2 destination_address_e        bit (8) ,
                    2 destination_address_f        bit (8) ,
                    2 source_address_a             bit (8) ,
                    2 source_address_b             bit (8) ,
                    2 source_address_c             bit (8) ,
                    2 source_address_d             bit (8) ,
                    2 source_address_e             bit (8) ,
                    2 source_address_f             bit (8) ,
                    2 type_field_a                 bit (8) ,
                    2 type_field_b                 bit (8) ,
                    2 data (238)                   char(1) ,
                    2 crc_msb                      bit (8) ,
                    2 crc_upper_middle_byte        bit (8) ,
                    2 crc_lower_middle_byte        bit (8) ,
                    2 crc_lsb                      bit (8) ,


               originator bit (8),
               rcvd_packet_type bit (8) ,
               copy_ie_register bit (8) ,
               (i,j,k) fixed bin (15),
               reg_value bit (8) .
               operation bit (8) ,
          border (80) char (1) static initial ((80)'-') ,

               /*   Modules external to this module */

               write_io_port entry (bit (8) ,bit (8)) ,
               read_io_port  entry (bit (8) ,bit (8)) ,
               initialize_cpu_interrupts     entry,
               enable_cpu_interrupts         entry,
               disable_cpu_interrupts        entry,
               write_bar entry (pointer);

               /*   end module listing  */
```

```
                *replace
/*  codes specific to the Intel 8259a Programmable
                    Interrupt Controller (PIC)      */

                    icw1_port_address           by 'c0'b4,
/* note that */     icw2_port_address           by 'c2'b4,
/* icw2,icw4,*/     icw4_port_address           by 'c2'b4,
/* and ocw   */     ocw_port_address            by 'c2'b4,
     /* use same  */
     /* port addr */
                         /* note: icw ==> initialization
                                          control
                                          word

                                  ocw ==> operational
                                          command
                                          word        */


                         icw1                   by '13'b4,

                       /* single PIC configuration, edge
                             triggered input              */

                    icw2                        by '40'b4,

                 /* most significant bits of vectoring
                              byte; for an interrupt 5,
                           the effective address will be
                          (icw2 + interrupt #) * 4 which
                             will be (40 hex + 5) * 4 =
                                114 hex                  */

                     icw4                       by '0f'b4,

                          /* automatic end of interrupt
                          and buffered mode/master     */
                     ocw1                       by 'df'b4,

                     /* unmask interrupt 5 (bit 5) and
                        mask all others                 */

                          /* end 8259a codes */

                     clearscreen              by '^z',
                     cluster0                 by '00'b4,
                     cluster1                 by '01'b4,
                     mds                      by '05'b4,
                     await_packet             by '00'b4,
                     packet_received          by '01'b4,
                     message_type             by '01'b4;
```

```
                  /* include constants specific to the RI3012
                  board                                          */

                      %include 'ri3012.dcl';

/***********************************************************************/

                          /*  Main Body */



             put list (clearscreen);
             put skip;
             put edit ((border (i) do i = 1 to 80)) (a);
             operation = await_packet;
         call read_io_port (command_status_register.reg_value);
             call initialize_pic;
             call initialize_cpu_interrupts;
             call perform_command (go_online);
             copy_ie_register = receive_block_available;
             call system_process;
             call perform_command (reset);
             put skip (3);
             put edit ((border (i) do i = 1 to 80)) (a);
             put skip (2);

           /* end main body */

/***********************************************************************/



           initialize_pic:    procedure;

               DECLARE

                   write_io_port entry (bit (8), bit (8));

                 call write_io_port (icw1_port_address,icw1);
                 call write_io_port (icw2_port_address,icw2);
                 call write_io_port (icw4_port_address,icw4);
                 call write_io_port (ocw_port_address,ocw1);

                 end initialize_pic;

/***********************************************************************/


           perform_command:        procedure (command);

                 DECLARE
```

```
                         command bit (8),
                         reg_value bit (8),
                         srf bit (8),
                        write_io_port entry (bit (8),
                                        bit (8)),
                        read_io_port  entry (bit (8),
                                        bit (8));

                   /* end declarations */

                   srf = '0'b4;
          call write_io_port (command_register,command);
                   do while ((srf & '01'b4) = '00'b4);
               call read_io_port (interrupt_status_reg,
                                        srf);
                   end;  /* do while */
            call read_io_port (command_status_register,
                                  reg_value);
                   if (reg_value > '01'b4) then
                   do;
            /* not (SUCCESS or SUCCESS with Retries) */

         put skip edit ('*** ETHERNET Board Failure ***')
                                    (col(20),a);
                        stop;
                   end; /* itd */


            end perform_command;




/*********************************************************************************/


         transmit_packet: procedure (cluster,packet_type)
                      external;


            DECLARE
                       cluster bit (8),
                       packet_type bit (8),
                       srf bit (8),
                       reg_value bit (8),
                       write_io_port entry (bit (8),
                                        bit (8)),
                       read_io_port entry (bit (8),
                                        bit (8)),
                   enable_cpu_interrupts            entry,
                   disable_cpu_interrupts           entry,
                                 write_bar entry  pointer ;









                    /* begin */
                    call disable_cpu_interrupts;
              do while (copy_ie_register = transmit_dma_done
                                          !
                       copy_ie_register = receive_dma_done);

                         call enable_cpu_interrupts;
              do while (copy_ie_register = transmit_dma_done
                                          !
                       copy_ie_register = receive_dma_done);
                         end;
                         call disable_cpu_interrupts;
                    end;
                copy_ie_register = disable_ni3210_interrupts;
                call write_io_port(interrupt_enable_register,
                                  disable_ni3210_interrupts);
                    srf = '0'b4;
                    if (cluster = cluster2) then
                    do;
          transmit_data_block.destination_address_5 = '[illegible]'b4;
          transmit_data_block.destination_address_6 = '[illegible]'b4;
                    end;

                    else     /* cluster = cluster1 */
                    do;
          transmit_data_block.destination_address_5 = '[illegible]'b4;
          transmit_data_block.destination_address_6 = '[illegible]'b4;
                    end;  /* else do */
                    if (packet_type = message_type) then
            transmit_data_block.type_field_a = message_type;
                 call write_bar (addr(transmit_data_block));
      call write_io_port(high_byte_count_reg,'0'b4);/*240 bytes*/
               call write_io_port(low_byte_count_reg,'F0'b4);
                    copy_ie_register = transmit_dma_done;
                call write_io_port(interrupt_enable_register,
                                          transmit_dma_done);
                    call enable_cpu_interrupts;
            do while (copy_ie_register = transmit_dma_done);
                 end;   /* loop until the interrupt handler
                             takes care of the TDD interrupt -
                                it sets IE_REG to 4 */
                    call write_io_port(command_register,
                                          load_and_send);
                    do while ((srf & '01'b4) = '00'b4);
                call read_io_port(interrupt_status_reg, srf);
```

```
                    end;    /* do while */
     call read_io_port(command_status_register,reg_value);

                end transmit_packet;

/*********************************************************************

                HL_interrupt_handler: procedure external;



                /* This routine is called from the low level
                 8086 assembly language interrupt routine. */

                    DECLARE

                            write_io_port entry (bit (8),
                                                  bit (8)),
                            read_io_port entry (bit (8),
                                                 bit (8)),
                        enable_cpu_interrupts            entry,
                        disable_cpu_interrupts           entry,
                            write_bar entry (pointer);

                    /*  begin   */
                    call disable_cpu_interrupts;
               call write_io_port(interrupt_enable_register,
                                  disable_ni3010_interrupts);
             if (copy_ie_register = receive_block_available)
                    then do;
                 call write_bar (addr(receive_data_block));

/* 260 bytes */
             call write_io_port(high_byte_count_reg,'01'b4);
              call write_io_port(low_byte_count_reg,'04'b4);

                        /* initiate receive DMA */

                      copy_ie_register = receive_dma_done;
               call write_io_port(interrupt_enable_register,
                                              receive_dma_done);

                    end;    /* do */
                    else
              if (copy_ie_register = receive_dma_done) then
                        do;
                            operation = packet_received;
               copy_ie_register = receive_block_available;
               call write_io_port(interrupt_enable_register,
                                     receive_block_available);
                        end;  /* if then do */
                    else
                  if  copy_ie_register = transmit_int_done
                         then do;
              copy_ie_register = receive_block_available;
              call write_io_port (interrupt_enable_register,
                                  receive_block_available);

                         end;   /* if then do */
               end HL_interrupt_handler;

/*************************************************************

        system_process: procedure;

                DECLARE

                        p pointer,
                convert_to_binary fixed bin (7) based (p),
                        tvar char (40) varying,
                        i fixed bin (15),
                write_io_port entry (bit (8),bit (8));

                copy_ie_register = receive_block_available;
             call write_io_port (interrupt_enable_register,
                                  receive_block_available);
      put skip list ('AWAITING NETWORK COMMUNICATIONS...');
                put skip;
                originator = '00'b4;
                do while ('1'b);
                   if (operation = packet_received) then
                        do;
                            call disable_cpu_interrupts;
       if (receive_data_block.type_field_a = message_type)
                            then do;
                        rcvd_packet_type = message_type;
   if  originator ^= receive_data_block.type_field_b) then
                        do;
              originator = receive_data_block.type_field_b;

                 put skip(3) edit ('Terminal ', originator,
                           ' sent the following message: ')
                                                (a,b4(2),a);
                                 i = 2;
                     p = addr (receive_data_block.data(1));
                   do while (i <= (convert_to_binary - 1));
               put edit (receive_data_block.data(i)) (a(1));
                                   i = i + 1;
                                end;
                        end; /* if then do */
                  transmit_data_block.type_field_b = mts;
              if (originator = 'Ø1'b4 | originator = 'Ø2'b4
                | originator = 'Ø3'b4 | originator = 'Ø4'b4)
                                            then
                                            do;

                                        if   originator = 'Ø1'b4  then
                  tvar = 'Terminal 1 message was received!';
                                        else
                                            if (originator = 'Ø2'b4) then
                  tvar = 'Terminal 2 message was received!';
                                                 else
                                            if (originator = 'Ø3'b4) then
                    tvar = 'Terminal 3 message was received!';
                                                 else
                                            if (originator = 'Ø4'b4) then
                    tvar = 'Terminal 4 message was received!';

                                        transmit_data_block.data = tvar;

                                            end;
                                            operation = await_packet;
                              end;
                              call enable_cpu_interrupts;
                              call transmit_packet(clusterØ, [illegible]);
                         end;
                   end;   /* do while forever */

          end system_process;

/***************************************************************************

end; /* procedure r5mod */
```

APPENDIX I

THE MODULE LISTING OF "ETHER.A86"

```
;Prog Name          :Ether.a86
;Date               :25 May 1984
;Written by         :Izzet Percinler
;For                :Thesis
;Advisor            :Professor Kodres
;Purpose            :Reinitializes common memory to the point
;                    at which the first user of the Ethernet
;                    services will now do a full reinitializat.
;                    of synchronization variables and also bring
;                    the NI3010 board on line.


        cseg

        mov     bx, 0e000h
        mov     es, bx
        mov     enetstart, 0
        mov     dl, 0                           ; release memory
        mov     cl, 0
        int     0e0h

        eseg

        org 5300h

        enetstart rb 1

        end
```

APPENDIX G

THE MODULE LISTING OF "TSTETHER.PLI"

```
tstether:        procedure options (main);

   /*    Date:                  1 June 1984

         Programmer:            Izzet Percinler

         Module  Function:      This module is a modified
                                version of "remote.pli" and
                                is designed to function as
                                a test program of the Ethernet
                                communications software to
                                demonstrate and analyze the
                                speed of data transferred over
                                Ethernet.




                DECLARE

                      1     transmit_data_block static,

                         2 destination_address_a
                              bit (8) initial ('02'b4),
                         2 destination_address_b
                              bit (8)initial  ('07'b4),
                         2 destination_address_c
                              bit (8)initial  ('01'b4),
                         2 destination_address_d
                              bit (8)initial ('2C'b4),
                         2 destination_address_e
                              bit (8) ,
                         2 destination_address_f
                              bit (8) ,
                         2 type_field_a
                              bit (8) ,
                         2 type_field_b
                              bit (8),
                         2 data  char (238) varying ,
                              .
                              .

                     1 receive_data_block,
                        2 frame_status              bit (8),
                        2 null_byte                 bit (8),
                        2 frame_length_lsb          bit (8),
                        2 frame_length_msb          bit (8),
                        2 destination_address_a     bit (8),
                        2 destination_address_b     bit (8),

                        2 destination_address_c     bit (8),
                        2 destination_address_d     bit (8),
                        2 destination_address_e     bit (8),
                        2 destination_address_f     bit (8),
                        2 source_address_a          bit (8),
                        2 source_address_b          bit (8),
                        2 source_address_c          bit (8),
                        2 source_address_d          bit (8),
                        2 source_address_e          bit (8),
                        2 source_address_f          bit (8),
                        2 type_field_a              bit (8),
                        2 type_field_b              bit (8),
                        2 data (238)               char (1),
                        2 crc_msb                   bit (8),
                        2 crc_upper_middle_byte     bit (8),
                        2 crc_lower_middle_byte     bit (8),
                        2 crc_lsb                   bit (8),


                    tvar char (234) varying,
                    terminal_service bit (8),
                    packet_type bit (8) ,
                    copy_ie_register bit (8) ,
                    (i,j,k) fixed bin (15),
                    reg_value bit (8) ,
               border (80) char (1) static initial ((80) '-'),
                    enet_init char (4),
                    user_count bit (8),
                    write_io_port entry (bit (8), bit (8)),
                    read_io_port  entry (bit (8), bit (8)),
                    move_to_lm entry (bit(16),pointer,
                                                fixed bin (15)),
                    move_to_cm entry (pointer, bit (16),
                                                fixed bin (15)),
                    initialize_cpu_interrupts         entry,
                    enable_cpu_interrupts             entry,
                    disable_cpu_interrupts            entry,
                    clear_ready_flag                  entry,
                    initsync                          entry,
                    set_ready_flag                    entry,
                    increment_user_count              entry,
                    decrement_user_count              entry,
                    write_bar entry (bit(16));
```

```
              /*    end module listing */





              %replace

      /*  codes specific to the Intel 8259A Programmable
                        Interrupt Controller (PIC)          */

                              icw1_port_address    by [illegible],
          /* note that */  icw2_port_address    by [illegible],
          /* icw2,icw4,*/  icw4_port_address    by [illegible],
          /* and ocw    */  ocw_port_address     by [illegible],
          /* use same   */
          /* port addr */

                       /* note: icw ==> initialization
                                         control
                                         word

                                   ocw ==> operational
                                         command
                                         word        */


                            icw1                   by '13'h,

                       /* single PIC configuration, edge
                            triggered input              */

                            icw2                   by '40'h,

                   /* most significant bits of vectoring
                              byte; for an interrupt 5,
                           the effective address will be
                          (icw2 + interrupt #) * 4 which
                              will be (40 hex + 5) * 4 =
                                  114 hex                 */

                             icw4                  by [illegible],

                            /* automatic end of interrupt
                            and buffered mode/master      */

                             ocw1                  by '9f'h,
                                    /* unmask interrupt 5 (bit 5) and
                                       d (bit 6), mask all others  */

                                         /* end 8259a notes */


                                         cluster0            by '00'b4,
                                         cluster1            by '01'b4,
                                         terminal_1          by '01'b4,
                                  message_type               by '01'b4,
                                terminal_service_request     by '01'b4,
                                 await_enet_access           by '02'b4,
                                         complete            by '00'b4,
                                         in_progress         by '01'b4,
                                         rts                 by '[illegible]'b4,
                                         not_ready           by '[illegible]'b4,
            /* ADM-3a specific */        clearscreen         by '[illegible]',
                                      addr_rcv_pkt_cm     by '[illegible]'b4,
                                      addr_xmit_pkt_cm    by '[illegible]'b4,
                                     status_addr          by '[illegible]'b4,
                                     count_addr           by '[illegible]'b4;

                         /* include constants specific to the NI3010
                          board                                      */

                              %include 'ni3010.dcl';

/*****************************************************************************

                                 /*  Main Body */



                   put list (clearscreen);
                   put skip;
                   put edit ((border (i) do i = 1 to 80)) (a);
                   put skip (2) list ('          WELCOME TO THE [illegible]');
                   put skip (2);
                   call initialize_pic;
                   call initialize_cpu_interrupts;
                   terminal_service = await_enet_access;
               call move_to_lm(enet_status_addr, addr(enet_init), 4);

                   if (enet_init ^= 'enet') then
                   do;
                     user_count = '00'b4;
                     enet_init = 'enet';
                call move_to_cm (addr(enet_init),enet_status_addr,4);
               call move_to_cm (addr(user_count), user_count_addr,1);
               call read_io_port (command_status_register,reg_value);
                     call perform_command (reset);
               call read_io_port (command_status_register,reg_value);
            call perform_command (go_online);
            call clear_ready_flag;
            call initsync;
          end;
          call increment_user_count;
          copy_ie_register = receive_block_available;
          call user_process;
          call write_io_port(ocw_port_address, 'bf'b4);
/* necessary actually only for iSBC1 as described in Perry's
thesis */

          call decrement_user_count; /* on way out */
          put skip (3);
          put edit ((border (i) do i = 1 to 27)) (a);
          put skip (2);

        /* end main body */

/*****************************************************************

               initialize_pic:    procedure;

             DECLARE

                    write_io_port entry (bit (8), bit (8));

               call write_io_port (icw1_port_address,icw1);
               call write_io_port (icw2_port_address,icw2);
               call write_io_port (icw4_port_address,icw4);
               call write_io_port (ocw_port_address, ocw1);

               end initialize_pic;

/*****************************************************************

               perform_command:        procedure (command);

                    DECLARE
                         command bit (8),
                         reg_value bit (8),
                         srf bit (8),
                      write_io_port entry (bit (8),
                                          bit (8)),
                      read_io_port  entry (bit (8),
                                          bit (8));

                    /* end declarations */
                    srf = '0'b4;
         call write_io_port (command_register,command);
                    do while ((srf & '1'b4) = '0'b4);
             call read_io_port (interrupt_status_reg,
                                                    srf);
                    end;  /* do while */
          call read_io_port (command_status_register,
                                                reg_value);
                    if (reg_value ^= '01'b4) then
                    do;
          /* not (SUCCESS or SUCCESS with Retries) */

       put skip edit ('*** ETHERNET  Board Failure ***')
                                      (col(5),a);
                           stop;
                    end; /* if */


          end perform_command;




/******************************************************************************/


         transmit_packet: procedure  external;



            DECLARE
                       srf bit (8),
                       reg_value bit (8),
                       write_io_port entry  (bit (8),
                                             bit (8)),
                       read_io_port entry   (bit (8),
                                             bit (8)),
                    enable_cpu_interrupts            entry,
                    disable_cpu_interrupts           entry,
                       write_bar entry  (bit (16));





            /* begin */
            call disable_cpu_interrupts;
     do while (copy_ie_register = transmit_dma_done
                    copy_ie_register = receive_dma_done;

                     call enable_cpu_interrupts;
           do while  copy_ie_register = transmit_dma_done

                    copy_ie_register = receive_dma_done;
                     end;
                     call disable_cpu_interrupts;
                end;
            copy_ie_register = disable_ni3010_interrupts;
            call write_io_port(interrupt_enable_register,
                               disable_ni3010_interrupts);
                srf = '0'h;

                call write_bar(addr_xmit_packet);
call write_io_port(high_byte_count_reg,'0'h4); /* [illegible] */
           call write_io_port(low_byte_count_reg,'[illegible]');
                copy_ie_register = transmit_dma_done;
            call write_io_port(interrupt_enable_register,
                                           transmit_dma_done);
                call enable_cpu_interrupts;
          do while (copy_ie_register = transmit_dma_done);
              end;    /* loop until the interrupt handler
                           takes care of the DMA interrupt -
                              it sets IE_REG to 4 */
                call write_io_port(command_register,
                                       load_and_send);
                do while (srf & '01'h) = '0'h;
            call read_io_port(interrupt_status_reg, srf);
                end;    /* do while */
    call read_io_port(command_status_register,reg_val);

            end transmit_packet;

/*****************************************************************

            HL_interrupt_handler: procedure external;



            /* This routine is called from the low level
             8086 assembly language interrupt routine */

                DECLARE

                       write_io_port entry (bit (8),
                                            bit (8)),
                       read_io_port entry  (bit (8),
                                            bit (8)),
                    enable_cpu_interrupts           entry,
                    disable_cpu_interrupts          entry,
                             write_par entry (bit 16);

                      /*  begin   */


                 if (terminal_service = in_progress) then
                 do;
              if (copy_ie_register = receive_block_available)
                      then do;
                call write_io_port(interrupt_enable_register,
                                         disable_ni3010_interrupts);
                                    call write_par (addr_rcv_pkt_cm);
/* 260 bytes */
              call write_io_port(high_byte_count_reg,(01h));
               call write_io_port(low_byte_count_reg,(04h));

                               /* initiate receive DMA */

                            copy_ie_register = receive_dma_done;
                call write_io_port(interrupt_enable_register,
                                                 receive_dma_done);

                     end;    /* do */
                     else
               if (copy_ie_register = receive_dma_done) then
                          do;
                              call set_ready_flag;
                 /* informs a terminal that data is rdy */
               copy_ie_register = receive_block_available;
               call write_io_port(interrupt_enable_register,
                                          receive_block_available);

                          end;  /* if then do */
                     else
                  if (copy_ie_register = transmit_dma_done)
                          then do;
                copy_ie_register = receive_block_available;
               call write_io_port(interrupt_enable_register,
                                          receive_block_available);

                          end;    /* if then do */
                  end;

                end HL_interrupt_handler;


/*************************************************************** /


        user_process: procedure ;
```

```
     DECLARE

           p pointer,
     convert_to_binary fixed bin (7) based (p),
           terminal bit (8),
           packet_type bit (8),
           cluster bit (8),
           (i,j,k) fixed bin (15),
           service_response char (10) varying,
           destination_response char (8),
           move_to_lm entry (bit(16),pointer,
                                    fixed bin (15)),
           move_to_cm entry (pointer, bit (16),
                                    fixed bin (15)),
           data bit (8) static init ('00'b4),
       read_ready_flag entry returns (bit (8)),
           request entry,
           release entry,
           clear_ready_flag entry;

     i=0;
     put skip list ('Terminal number: ');
     get edit (terminal) (b4(2));
     transmit_data_block.type_field_b = terminal;
     service_response = 'continue';
     put skip;
     do while ('1'b);

transmit_data_block.destination_address_e = '[illegible]'b4;
transmit_data_block.destination_address_f = '4A'b4;
  transmit_data_block.type_field_a = message_type;
             tvar = 'Data packet sent';
             transmit_data_block.data = tvar;
    /* now must get the right to use the [illegible]
       resource multiplexed among all users.
       In reality the limited resource is the
     transmit packet template in common memory */

      call request;  /* Intel 8086 assembly lang.
                      routine - will loop infinit.
                        until my service number
                        is reached          */

             do while (i<50);
               data = not_ready;
               i = i + 1 ;
              terminal_service = in_progress;
  /* my turn!!! So, write the transmit packet
     to the template in common memory   */

    call move_to_cm (addr(transmit_data_block),
                      addr_xmit_pkt_cm,
                                          248);
                        call transmit_packet;

                        do while (data = not_ready);
                        data = read_ready_flag();
               /* 8086 routine that reads
               flag which ETHERNET communications
               handler sets to one when data is
                            available     */
                        end;

                /* data is ready in common memory -
            remote host has responded; so get data */

                     call move_to_lm (addr_rcv_buf_cm,
                           addr(receive_data_block),
                                      248);

                        call clear_ready_flag;
                        end;
                        i = 0;
                        put edit ('A') (a(1));
                      . terminal_service = complete;
           call release;  /* allows next user to be
                              serviced    */
                      /* display response on CRT */

              /* get type_field_t to determine what host
             is trying to communicate with this terminal
                and then create an appropriate response
                                         */

      end; /* do while */
    end; /* user_process */
end; /* procedure tstether */
```

LIST OF REFERENCES

1. Perry, M. L., *Logic Design of a Shared Disk System in a Multi-Micro Computer Environment*, M. S. Thesis, Naval Postgraduate School, June 1983

2. Candalor, M. B. , *Alteration of the CP/M-86 Operating System*, M. S. Thesis, Naval Postgraduate School, June 1983

3. Clark, D.D. , Pogran, K.T. , Reed, D.P. "An Introduction to Local Area Networks" *Tutorial Local Computer Networks*, 1981

4. Metcalfe, R. , Boggs, D. , "Ethernet: Distributed Packet Switching for Local Area Networks" , *Communications of ACM*, July 1976

5. Xerox Corporation, *The Ethernet - A Local Area Network: Data Link Layer and Physical Layer Specifications*, *Version 2.0* , November, 1982

6. Shoch J. F. , Dalal Y. K., Redall D. D. , Crane, R. C. "Evolution of the Ethernet Local Computer Network", *Computer*, August 1982

7. Interlan Corporation, *NI3010 Multibus Communications Controller Users Manual*, 1982

8. Xerox Corporation *The Ethernet - A Local Area Network:Data Link Layer and Physical Layer Specifications*, Version 1.0, September 1980

9. Stotzer, M. D. , *A Layered Communication System for Ethernet*, M. S. Thesis, Naval Postgraduate School, September, 1983

10. Kodres, U. R., "Processing Efficiency of a Class of Multicomputer Systems" , *Proceedings MIMI*, 1982

11. Almquist, T. V. and Stevens, David S. , *Alteration and Implementation of the CP/M-86 Operating System for a Multi-User Environment*, M. S. Thesis, Naval Postgraduate School, December 1982

12. Rowe, B. , *Adaptation of MCORTEX to the AEGIS Simulation Environment*, M. S. Thesis, Naval Postgraduate School, June 1984

13. Reed D. P. , Kanodia R .K., "Synchronization with Eventcounts and Sequencers" , *Communications of the ACM*, February 1979

BIBLIOGRAPHY

Digital Research, *Programmer's Utilities Guide for the CP/M Family of Operating Systems* Pacific Grove, September 1982.

Digital Research, *PL/I Language Programmer's Guide*, Pacific Grove, 1983.

Digital Research, *PI/I Language Reference Manual*, Pacific Grove, 1983.

Digital Research, *LINK-86 Operator's Guide*, Pacific Grove, 1982.

Digital Research, *CP/M-86 Operating System System Guide*, Pacific Grove, 1983.

Digital Research, *CP/M-86 Operating System User's Guide*, Pacific Grove, 1983.

Digital Research, *CP/M-86 Operating System Programmer's Guide*, Pacific Grove, 1983.

Hogan, T., *Osborne CP/M User-Guide*, Osborne/McGraw-Hill, 1982.

Intel Corporation, *iSBC 86/12A Single Board Computer Hardware Reference Manual*, 1976.

Intel Corporation, *ICS-80 Industrial Chassis Hardware Reference Manual*, 1979.

Miller, A. M. , *Mastering CP/M*, Sybex, 1983.

Zaks, R. , *The CP/M Handbook with MP/M*, Sybex,1980.

INITIAL DISTRIBUTION LIST

No. Copies

1. Defense Technical Information Center 2
Cameron Station
Alexandria, Virginia 22314

2. Library, Code 0142 2
Naval Postgraduate School
Monterey, California 93943

3. Prof. Uno R. Kodres 3
Code 52 KR
Department of Computer Science
Naval Postgraduate School
Monterey, California 93943

4. Department Chairman 1
Code 52 Hq
Department of Computer Science
Naval Postgraduate School
Monterey, California 93943

5. LCDR Ronald B. Kurth 1
Code 52 Kn
Department of Computer Science
Naval Postgraduate School
Monterey, California 93943

6. Prof. Mitchell F. Cotton 1
Code 62 Cc
Department of Electrical Engineering
Naval Postgraduate School
Monterey, California 93943

7. Prof. Richard McGonigal 1
Code 54 Mb
Department of Administrative Science
Naval Postgraduate School
Monterey, California 93943

8. Lt. David J. Brewer 1
Department of Computer Science
CS 31
Naval Postgraduate School
Monterey, California 93943

9. RCA AEGIS Data Repository 1
RCA Corporation
Government Systems Division
Mail Stop 127-327
Moorestown, New Jersey 08057

10. Library (Code E33-05) 1
Naval Surface Warfare Center
Dahlgren, Virginia 22449

11. Daniel Green (Code N20E) 1
Naval Surface Warfare Center
Dahlgren, Virginia 22449

12. Dr. M.J. Gralia 1
Applied Physics Laboratory
Johns Hopkins Road
Laurel, Maryland 20707

13. Dana Small 1
Code 8242, NOSC
San Diego, California 92152

14. Genelkurmay Baskanligi 2
OBID
Bakanliklar, Ankara, Turkey

15. Kara Kuvvetleri Komutanligi 2
Pl. Pren. Bsk. KOKOBI
Yucetepe, Ankara, Turkey

16. Bnb. Izzet PERCINLER 3
Kara Kuvvetleri Komutanligi
Pln. Pren. Bsk. KOKOBI
Yucetepe, Ankara, Turkey

18. Orta Dogu Teknik Universitesi 1
Bilgisayar Bolum Baskani
Ankara, Turkey

19. Prof. Halim Dogrusoz 1
Orta Dogu Teknik Universitesi
Yoneylem Arastirma Bolum Baskani
Ankara, Turkey

20. Hacettepe Universitesi 1
Bilgisayar Bolum Baskani
Ankara, Turkey

21. Sn. Erol Aksoy 1
Bogazici Universitesi
Bilgisayar Bolum Baskanligi
Ogretim Uyesi
Istanbul, Turkey

22. Bogazici Universitesi 1
Bilgisayar Bolum Baskani
Istanbul, Turkey

23. Istanbul Teknik Universitesi 1
Bilgisayar Bolum Baskani
Istanbul, Turkey

210541

Thesis
P3342 Percinler
c.1 Multiplexing Ethernet
in a multi-user CP/M-86
system.